The Journal of Supercomputing (2025) 81:772
https://doi.org/10.1007/s11227-025-07063-7

Analyzing performance portability for a SYCL implementation of the 2D shallow water equations

Markus Büttner[1] · Christoph Alt[2,3] · Tobias Kenter[2] · Harald Köstler[3] · Christian Plessl[2] · Vadym Aizinger[1]

Accepted: 12 February 2025

Abstract

SYCL is an open standard for targeting heterogeneous hardware from C++. In this work, we evaluate a SYCL implementation for a discontinuous Galerkin discretization of the 2D shallow water equations targeting CPUs, GPUs, and also FPGAs. The discretization uses polynomial orders zero to two on unstructured triangular meshes. Separating memory accesses from the numerical code allow us to optimize data accesses for the target architecture. A performance analysis shows good portability across x86 and ARM CPUs, GPUs from different vendors, and even two variants of Intel Stratix 10 FPGAs. Measuring the energy to solution shows that GPUs yield an up to 10x higher energy efficiency in terms of degrees of freedom per joule compared to CPUs. With custom designed caches, FPGAs offer a meaningful complement to the other architectures with particularly good computational performance on smaller

✉ Markus Büttner
markus.buettner@uni-bayreuth.de

Christoph Alt
christoph.alt@uni-paderborn.de

Tobias Kenter
kenter@uni-paderborn.de

Harald Köstler
harald.koestler@fau.de

Christian Plessl
christian.plessl@uni-paderborn.de

Vadym Aizinger
vadym.aizinger@uni-bayreuth.de

1 Chair of Scientific Computing, University of Bayreuth, Universitätsstraße 30, 95447 Bayreuth, Germany

2 Paderborn Center for Parallel Computing, Paderborn University, Warburger Str. 100, 33098 Paderborn, Germany

3 Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg, Cauerstraße 11, 91058 Erlangen, Germany

Published online: 23 April 2025

meshes. FPGAs with High Bandwidth Memory are less affected by bandwidth issues and have similar energy efficiency as latest generation CPUs.

Keywords Performance portability · SYCL · Shallow water equations · Discontinous Galerkin method · GPU programming · FPGA

1 Introduction

The hardware used for scientific computing has become more diverse in recent years. In the TOP500 lists of supercomputers from November 2024, all but one system in the top 10 use GPUs from AMD, Intel, and Nvidia, and the ARM architecture-based Fugaku system is now on sixth position. Additionally, Nvidia's new Grace CPU (also ARM based) now occupies five out of the first 10 places in the Green 500 list. Furthermore, researchers are interested in utilizing other accelerator technologies like FPGAs, quantum processors, or AI chips.

Scientific codes such as ocean, weather, and climate models need to adapt to these architectures in order to increase spatial resolution, lengthen simulated time intervals, add physical features, and thereby, ultimately, improve the accuracy of models and predictions. Due to long development cycles and lifetimes of such codes, it is important to identify, with sufficiently specific case studies, fitting abstractions and programming models that allow to make good use of the different available HPC hardware. Even when functional portability is given, optimizing performance for each architecture is often a time-consuming process, which may have to be repeated for each new target hardware. Performance portable codes try to minimize this effort. Although specifics for a suitable performance portability metric are still in discussion [1–4], its general goal is the ability to achieve good architectural or application efficiency on a wide range of target architectures while maintaining, at the same time, developer productivity via a coherent code base with minimal code specialization for different hardware.

One avenue toward performance portability are libraries and domain-specific languages (DSLs), which hide architecture-specific optimizations in the underlying libraries or DSL and implement application functionality in codes on top of this layer following the separation-of-concerns paradigm. However, libraries may be limited in their feature sets and the interfaces they provide, while DSLs often rely on a small number of specialized developers and maintainers to add new optimization strategies, target architectures, or even maintain functional support. Considering this, codes with offloading directives via OpenACC, OpenMP, or cross-platform parallel programming standards like Kokkos [5], OpenCL, or SYCL are the preferred alternative for many projects – not least due to their closeness in syntax and features to high level programming languages such as C or C++.

SYCL builds upon the syntax of modern C++ and was conceptualized as a less verbose (via single-source host and device code) and more feature-rich evolution of OpenCL for heterogeneous device programming. SYCL aims to provide platform portability for a wide range of target architectures; previous studies [6–8] have shown that performance comparable to vendor-specific programming models like CUDA is possible with SYCL, though individual results differ case by case for different applications or benchmark settings. First projects [9, 10] are implementing performance portable applications using the SYCL standard.

This work describes and evaluates the implementation of a SYCL-based shallow water solver for CPUs, GPUs, and FPGAs aiming for performance portability. The utilized Intel oneAPI and AdaptiveCpp [11, 12] compilers allow to execute the code on a wide range of hardware architectures. To obtain good performance, some of the necessary optimizations are customized for specific target architectures, but such specializations can be isolated from the common numerical code. We use the resulting application to compare performance and energy efficiency of different architectures for this type of workload. This work extends the previous conference publication "Enabling Performance Portability for Shallow Water Equations on CPUs, GPUs, and FPGAs with SYCL" [13] by providing a more detailed performance analysis and including ARM-based CPUs as well as FPGAs with High-Bandwidth-Memory (HBM2). Furthermore, we complement the performance results by an energy efficiency analysis across a heterogeneous set of different architectures.

The remainder of the paper is structured as follows: We start with an overview of related work in Sect. 2. Section 3 explains the modal discontinuous Galerkin discretization employed by our implementation together with the two test cases used in this study. Section 4 discusses the SYCL implementation of the discretization as well as the employed optimizations and specializations for memory accesses on different architectures. The main evaluation and analysis in Sect. 5 starts with a validation of the implementation and a general performance overview on different platforms before exploring the individual architectures, CPUs, GPUs, and FPGAs in more detail to understand performance implications of architectures, code choices, and compilers. Thus Sect. 5.3.3 illustrates the impact of one specific optimization over the code version from [13] on performance on different hardware. Evaluation of the energy efficiency and case studies with further target platforms conclude the analysis before we close the paper with a summary and outlook in Sect. 6.

2 Related work

While SYCL does not guarantee performance portability, providing only abstractions to write platform portable code, it was previously shown [6–8] that performance can be competitive to OpenCL or CUDA/HIP. The GROMACS [14] molecular dynamics software adopted SYCL as portable GPU acceleration API for AMD, Intel, and Nvidia GPUs. Even though for Nvidia GPUs the CUDA backend is still the recommended GROMACS setup,[1] the results of [8] show that either of the two versions can be faster depending on the dataset or benchmark.

[9] used SYCL to implement a performance portable cosmological N-body structure formation code for AMD, Intel and Nvidia GPUs, and [10] ported the MadGraph algorithm from OpenMP/CUDA to SYCL and obtained comparable or better performance. [15] compared Kokkos, SYCL and CUDA for the Milc-Dslash benchmark

[1] https://manual.gromacs.org/documentation/2024.4/install-guide/index.html.

(simulating lattice quantum chromodynamics) and found that Kokkos and SYCL yield comparable performance, close to the native CUDA implementation. However, they also note that at the time of writing, Kokkos was more readily available (and thus portable) than SYCL. In particular, they were not able to include SYCL results for an AMD GPU, neither with one of the first releases of the Intel oneAPI compiler or its open-source LLVM branch nor with hipSYCL [16] that now has become AdaptiveCpp [11, 12]. Four years later, our work uses both of these combinations without problem, as already shown before by Reguly [7], and in one of the combinations by several of the other referenced studies [6, 9, 10].

[9] quantify the performance portability of their SYCL code with the *application efficiency* variant of the Φ metric according to Pennycook et al. [17, 18] as 0.96. Φ metrics can be useful to compare the performance portability of different implementations on the same set of target platforms. Particularly the metric based on application efficiency relies on optimized reference implementations, ideally with lower level programming models (there are CUDA and HIP variants in [9]; Marowka [19] discusses further limitations), whereas metrics based on *architectural efficiency* are hard to interpret beyond a comparison of the same application on the same set of platforms, as in [7]. Furthermore, there are ongoing discussions [1–4] about the right formulation of performance portability metrics with regard to averaging and dealing with unsupported platforms. In this work, we focus on a qualitative understanding of performance portability with detailed analysis of application performance and how it relates to platform characteristics.

Regarding SYCL codes with FPGA support, Weckert et al. [20] recently presented a SYCL version of the Altis [21] GPGPU benchmark, which they first optimized for GPU execution before subsequently adding an FPGA version – however without aiming for a unified code base between the two. Executing a total of 12 benchmarks on three different data sizes each, one of their findings connects to an observation in our work. In seven of their benchmarks, FPGAs show best speedups for the smallest data set, from 1.77× – 13.12× relative to the 6-core CPU reference. The speedups shrink for the largest data set to 0.62× – 2.34×, attributed to bandwidth limitations on the FPGA platform, which are also clearly visible in our results (cf. Sections 5.5 and 5.5.1). [22] use SYCL and oneAPI to target Intel CPUs, GPUs, and FPGAs for an ultrasound beamforming application. For the FPGA target, they also conduct a complete rewrite of the kernels and additionally exploit task level parallelism by pipelining data through four subsequent kernels. A similar approach was taken for the first FPGA variant of our shallow water code with OpenCL [23]. However, the results of the current work show that similar performance characteristics on FPGA can be obtained with a single kernel, where only a limited set of specializations is employed and decoupled from the numerical code. Other works [24–27] show promising FPGA performance with SYCL and oneAPI but use other implementations for performance comparisons on other targets and don't aim for platform or performance portability yet. All four approaches rely heavily on creating different specialized on-chip memory structures. Though our work also creates additional memory subsystems on the FPGA targets (see Sects. 4.2.2 and 4.2.3), it is unclear if any of the other projects could be refactored into a similarly unified portable codebase as presented in the current work.

There are several other implementations of the shallow water equations. In what might be the most similar work to ours, [28] also implemented the SWE on unstructured meshes. They target CPUs and GPUs and use OCCA as performance portability layer for OpenCL, CUDA, and OpenMP backends. Interestingly, similarly to our results in Sect. 5.3, they already observed 10 years ago effective vectorization over work items in the OpenCL backend and corresponding differences to OpenMP performance. [29] implemented a finite volume scheme on Cartesian grids for the SWE on CPUs and GPUs with Kokkos demonstrating good scalability on up to 256 GPUs. Further implementations by [30] and [31] also use finite volumes on a Cartesian grid but rely on CUDA for GPU acceleration. ExaHyPE [32] is a general framework for solving hyperbolic PDEs and can also be used to solve the shallow water equations by the discontinuous Galerkin method. Again, they use adaptively refined Cartesian grids and SIMD instructions in combination with MPI or Intel TBB for parallelization on CPUs. GPU support in ExaHyPE with SYCL was studied in a recent work by [33]. An implementation based on algorithmic skeletons was presented in [34], again with a finite volume discretization on Cartesian grids and only on CPUs. Conceptually, SWE on Cartesian grids can use more regular memory accesses than on unstructured meshes thus reaching higher effective bandwidth and – depending on the arithmetic intensity – higher computational throughput. On the other hand, unstructured meshes can be fitted better to irregular coastline shapes and thus can allow for coarser geometries and less overall computational effort. However, for wider acceptance of methods on unstructured meshes, high performance and portability need to be shown, to which we contribute with this work.

Faghih-Naini et al. [35] used the code-generation framework "ExaStencils" to solve the SWE using a quadrature-free DG method on block-structured grids, targeting CPUs through OpenMP and MPI as well as GPUs with CUDA. Based on [35] and exploiting the structured grid format to use a stencil formulation with line buffers and temporal parallelism, Alt et al. [36] demonstrated a first FPGA implementation supporting much larger grids than in [23, 37], albeit without the high performance for small grids that was achieved in this work. An earlier FPGA design [38] based on the nodal DG method for Maxwell's equations in 3D is also supporting arbitrary grid sizes via off-chip global memory but suffers less from bandwidth limitations due to the higher polynomial orders suitable for Maxwell's equations. Later, Gourounas et al. [39] have generalized such an architecture around the idea of element processors for nodal DG that can be customized on FPGA for different equations and evaluate it with an elastic wave application.

3 Discontinous Galerkin method for the 2D shallow water equations

The 2D shallow water equations model the depth-integrated circulation in free surface water bodies such as lakes, seas, and oceans. They are also widely used as a prototype for dynamical cores of three-dimensional models of ocean and atmosphere [40]. The conservative form of the SWE on a 2D domain Ω is formulated as follows:

$$\partial_t \xi + \nabla \cdot \boldsymbol{q} = 0, \tag{1}$$

$$\partial_t \boldsymbol{q} + \nabla \cdot \left(\boldsymbol{q}\boldsymbol{q}^T / H \right) + \tau_{\mathrm{bf}} \boldsymbol{q} + \begin{pmatrix} 0 & -f_c \\ f_c & 0 \end{pmatrix} \boldsymbol{q} + gH\nabla\xi = \boldsymbol{F}, \tag{2}$$

where $\boldsymbol{q} := (U, V)^T$ is the depth-integrated horizontal velocity and ξ the water elevation with respect to some datum (e.g., the mean sea level). For h_b, the bathymetric depth respective the same datum, $H = h_b + \xi$ gives the total water depth. The other terms are: f_c the Coriolis coefficient, g the gravitational acceleration, and τ_{bf} the bottom friction coefficient; $\boldsymbol{F} = (F_x, F_y)^T$ contains forcings such as gradients of atmospheric pressure and tidal potential. Defining $\boldsymbol{c} := (\xi, U, V)^T$, a compact form of Eqs. 1–2 is given by

$$\partial_t \boldsymbol{c} + \nabla \cdot \boldsymbol{A}(\boldsymbol{c}) = \boldsymbol{r}(\boldsymbol{c}), \tag{3}$$

where

$$\boldsymbol{A}(\boldsymbol{c}) = \begin{pmatrix} U & V \\ \frac{U^2}{H} + \frac{g(H^2 - h_b^2)}{2} & \frac{UV}{H} \\ \frac{UV}{H} & \frac{V^2}{H} + \frac{g(H^2 - h_b^2)}{2} \end{pmatrix}, \tag{4}$$

$$\boldsymbol{r}(\boldsymbol{c}) = \begin{pmatrix} 0 \\ -\tau_{\mathrm{bf}} U + f_c V + g\xi\partial_x h_b + F_x \\ -\tau_{\mathrm{bf}} V - f_c U + g\xi\partial_y h_b + F_y \end{pmatrix}. \tag{5}$$

The governing equations are complemented by the appropriate initial and boundary conditions. For $\boldsymbol{n}$, an exterior unit normal to the boundary of Ω, we set no normal flow at the *land* boundaries

$$\boldsymbol{q} \cdot \boldsymbol{n} = 0, \tag{6}$$

and specify the water elevation $\hat{\xi}$ at the *open sea* boundaries as

$$\xi = \hat{\xi}. \tag{7}$$

3.1 Discontinuous Galerkin discretization

The discretization of Eq. 3 uses the DG method first presented in [41]. We denote by Ω_e, $e \in \{1, \ldots, E\}$ a partition of $\Omega \subset \mathbb{R}^2$ into triangular elements, $\mathbb{P}^k(\Omega_e)$ the space of polynomials of degree k on Ω_e and utilize the standard notation $(\cdot, \cdot)_{\Omega_e}$ and $\langle \cdot, \cdot \rangle_{\partial\Omega_e}$ for the L^2-scalar products on elements and edges, respectively. We seek the finite-dimensional solution $\boldsymbol{c}_\Delta(t, \cdot) \in \mathbb{P}^k(\Omega_e)^3$ satisfying the semi-discrete formulation given on an element Ω_e by

$$\left(\partial_t \boldsymbol{c}_\Delta, \boldsymbol{\phi}_\Delta\right)_{\Omega_e} - \left(\boldsymbol{A}(\boldsymbol{c}_\Delta), \nabla\boldsymbol{\phi}_\Delta\right)_{\Omega_e} + \langle \hat{\boldsymbol{A}}(\boldsymbol{c}_\Delta, \boldsymbol{c}_\Delta^+, \boldsymbol{n}), \boldsymbol{\phi}_\Delta \rangle_{\partial\Omega_e} = \left(\boldsymbol{r}(\boldsymbol{c}_\Delta), \boldsymbol{\phi}_\Delta\right)_{\Omega_e} \tag{8}$$

for all test functions $\boldsymbol{\phi}_\Delta \in \mathbb{P}^k(\Omega_e)^3$. $\boldsymbol{n}$ is an exterior unit normal to $\partial\Omega_e$ and $\hat{\boldsymbol{A}}(\boldsymbol{c}_\Delta, \boldsymbol{c}_\Delta^+, \boldsymbol{n})$ denotes a numerical flux computed from the (discontinuous) values of $\boldsymbol{c}_\Delta$ on element Ω_e (without superscript) and its edge neighbor (superscript $^+$). On exterior domain boundaries $\boldsymbol{c}_\Delta^+$ is replaced by the specified boundary conditions. Our numerical scheme employs the local Lax-Friedrichs flux [42] given by

$$\hat{\boldsymbol{A}}(\boldsymbol{c}_\Delta, \boldsymbol{c}_\Delta^+, \boldsymbol{n}) = \frac{1}{2}\Big(\big(\boldsymbol{A}(\boldsymbol{c}_\Delta) + \boldsymbol{A}(\boldsymbol{c}_\Delta^+)\big) \cdot \boldsymbol{n} + \lambda(\boldsymbol{c}_\Delta - \boldsymbol{c}_\Delta^+)\Big), \tag{9}$$

where λ is the largest (in absolute value) eigenvalue of $\frac{\partial}{\partial \boldsymbol{c}}(\boldsymbol{A}(\boldsymbol{c}) \cdot \boldsymbol{n})$

$$\lambda(H, \boldsymbol{q}) = \left|\frac{\boldsymbol{q} \cdot \boldsymbol{n}}{H}\right| + \sqrt{gH}. \tag{10}$$

Denoting by $\boldsymbol{e}_j$ the j-th unit vector in $\mathbb{R}^3$, $\boldsymbol{c}_\Delta$ has the representation

$$\boldsymbol{c}_\Delta(t, \boldsymbol{x})|_{\Omega_e} = (\xi_\Delta, U_\Delta, V_\Delta)^T(t, \boldsymbol{x})|_{\Omega_e} = \sum_{j=1}^{3}\sum_{i=1}^{K(k)} c_{ei}^j \varphi_{ei}(\boldsymbol{x})\, \boldsymbol{e}_j, \tag{11}$$

where $\varphi_{ei}(\boldsymbol{x})$, $i = 1, \dots, K(k)$ are a basis of $\mathbb{P}^k(\Omega_e)$. In $\mathbb{R}^2$, the values of $K(k)$ are: $K(0)=1$, $K(1)=3$, $K(2)=6$. In our implementation, we use *modal* basis functions which are orthonormal with respect to the L^2-scalar product on Ω_e resulting in a unity mass matrix.

For the time step size Δt and the current time level $t_n = n\Delta t$, we write $\boldsymbol{C}^n$ for the global vector of degrees of freedom at time $t = t_n$. Then our explicit, strong-stability-preserving (SSP) Runge–Kutta time stepping scheme with s stages is given by [43]

$$\begin{aligned}
\boldsymbol{C}^{(0)} &= \boldsymbol{C}^n, \\
\boldsymbol{C}^{(k)} &= \sum_{i=0}^{k-1} \alpha_{ki}\boldsymbol{C}^{(i)} + \beta_k \Delta t\, \mathcal{L}\Big(\boldsymbol{C}^{(k-1)},\ t_n + \delta_k \Delta t\Big),\ k = 1, 2, \dots, s, \\
\boldsymbol{C}^{n+1} &= \boldsymbol{C}^{(s)},
\end{aligned} \tag{12}$$

where $\mathcal{L}(\cdot, \cdot)$ denotes the DG space discretization (the discrete representation of terms two, three, and four of Eq. 8). The first-order scheme is just the explicit Euler method (i.e., $\alpha_{01} = \beta_1 = 1$, $\delta_0 = 0$). The coefficients of the second-order method are as follows: $\alpha_{10} = \beta_1 = 1$, $\alpha_{20} = \alpha_{21} = \beta_2 = 1/2$, $\delta_0 = 0$, $\delta_1 = 1$. Finally, the third-order Runge–Kutta time integrator is specified by $\alpha_{10} = \beta_1 = 1$, $\alpha_{20} = 3/4$, $\alpha_{21} = \beta_2 = 1/4$, $\alpha_{30} = 1/3$, $\alpha_{31} = 0$, $\alpha_{32} = \beta_3 = 2/3$, $\delta_0 = 0$, $\delta_1 = 1$, $\delta_2 = 1/2$.

3.2 Reference implementation – UTBEST

The DG scheme outlined above was originally implemented as a hybrid FORTRAN77/C code called UTBEST [41], whose legacy FORTRAN part originated from a finite volume implementation of the SWE [44] and was retained to handle I/O and mesh management, while the DG discretization and all computationally intensive parts use C. The implementation was later ported to various other programming frameworks such as MATLAB [42], EXASTENCILS code generation framework [45], or OpenCL for FPGAs [23].

UTBEST utilizes unstructured triangular meshes with piecewise constant ($k = 0$, P0) (identical to cell-centered finite volumes), linear ($k = 1$, P1), and quadratic ($k = 2$, P2) DG polynomial spaces. The time discretization utilizes explicit strong stability preserving (SSP) Runge–Kutta methods described in (12) with the temporal discretization order chosen automatically to match the spatial discretization order. The solution algorithm (cf. Algorithm 1) contains the bulk of computation in the element and the edge integration kernels. The element kernel involves a loop over quadrature points on a triangle (one point for P0, four for P1, and seven for P2); the edge kernels iterate over 1D quadratures (one point for P0, two for P1, and three for P2) and call the Lax-Friedrichs flux function in each quadrature point.

Algorithm 1 Original algorithm of UTBEST with separate element and edge integration kernels

```
1:  while $t < t_1$ do
2:      Loop over Runge–Kutta stages:
3:      for stages of the Runge–Kutta method do
4:          Element loop:
5:          for element indices $e \in \{1, \dots, E\}$ do
6:              calculate element integrals for terms two, four in Eq. 8 on $\Omega_e$
7:          end for
8:          Loops over edges of different types:
9:          for interior edges do
10:             calculate edge integrals for term three in Eq. 8 and update both elements containing the edge
11:         end for
12:         for land edges do
13:             calculate edge integrals for term three in Eq. 8 and update the element containing the edge
14:         end for
15:         for open sea edges do
16:             calculate edge integrals for term three in Eq. 8 and update the element containing the edge
17:         end for
18:         calculate $\boldsymbol{C}^{(k)}$ for the next Runge-Kutta stage
19:         Element loop:
20:         for element indices $e \in \{1, \dots, E\}$ do
21:             perform minimum depth control on $\boldsymbol{c}_\Delta$
22:         end for
23:     end for
24:     $t \leftarrow t + \Delta t$
25: end while
```

In [23], an alternative projection approach has been developed: First, the L^2 projection of the local element solution onto the edges of the element is computed and stored in memory. The edge integrals are then computed element-wise. This enabled a better dataflow architecture in the FPGA design and is also used for the SYCL implementation presented in this work. Since only the local element solution is updated in this reformulation, parallelization over the elements becomes much easier, and only one global synchronization point is required after the projections have been computed.

3.3 Test cases

We use two realistic domains as test cases in this paper. Both domains, the Bight of Abaco and Galveston Bay, use similar parameters and boundary conditions, albeit the geometry of the Galveston Bay is more complex than the Bight of Abaco.

3.3.1 Tidal flow around Bahamas

The first test scenario considered in this paper models tide-driven flow at the Bahamas Islands and is described in [45]. Figure 1 (left) shows the domain geometry, bathymetry, boundary types, and locations of four observation stations monitoring the temporal evolution of surface elevation and depth-integrated velocity. For bottom friction, we use the standard quadratic friction law $\tau_{\mathrm{bf}} = C_f|\boldsymbol{q}|/H^2$ with coefficient $C_f = 0.009$. The constant Coriolis parameter is set to $3.19 \times 10^{-5}\ \mathrm{s}^{-1}$. The following tidal forcing with time t in hours is prescribed at the open sea boundary:

$$\hat{\xi}(t) = 0.075\cos\left(\frac{t}{25.82} + 3.40\right) + 0.095\cos\left(\frac{t}{23.94} + 3.60\right)$$

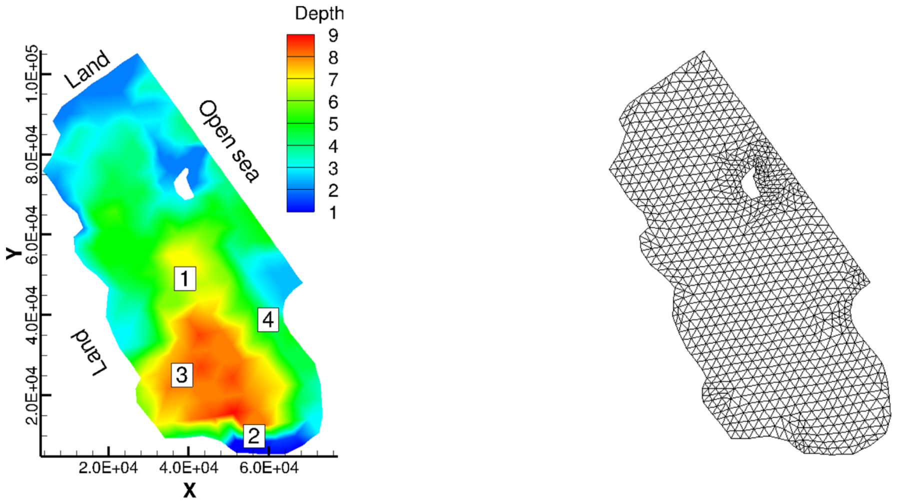


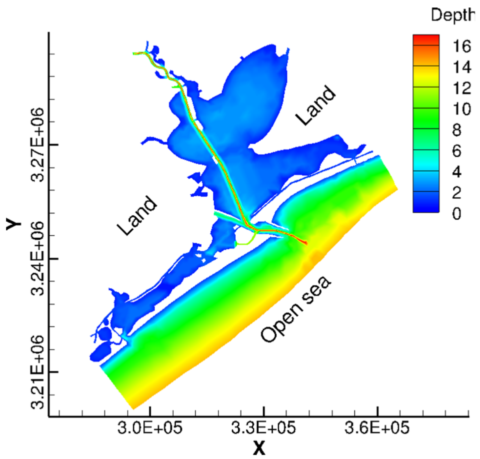
Fig. 1 Bahamas (Bight of Abaco from [46]): Bathymetry, boundary type, and observation stations (left), original triangular mesh with 1696 elements (right)

$$+ 0.1 \cos\left(\frac{t}{12.66} + 5.93\right) + 0.395 \cos\left(\frac{t}{12.42}\right) + 0.06 \cos\left(\frac{t}{12.00} + 0.75\right). \tag{13}$$

Figure 1 (right) illustrates the initial computational mesh with 1696 triangles. By subdividing each triangle via edge bisection into four similar triangles and linearly interpolating bathymetry we generated, based on this initial mesh, finer meshes with up to seven million elements for the performance studies. The time step used for our simulation runs depends on the mesh resolution and the spatial discretization order (P0, P1, or P2), while the simulated time interval was chosen depending on the mesh resolution and the hardware type to facilitate meaningful performance measurements.

3.3.2 Galveston Bay

As a second test case, we simulate tidal flow within the Galveston Bay (cf. Figure 2) for one day. The geometry and the bathymetry of the computational domain is much more complicated in this scenario than in the Bahamas simulation and includes 17 islands and a deep channel (Houston Ship Channel) within the bay. The open sea forcing imposed at the south-eastern boundary is the same as in the previous problem and given in (13), land boundary conditions were imposed at all other exterior boundaries. The quadratic bottom friction law with a constant coefficient $C_f = 0.004$ and a constant Coriolis parameter set to 7.07×10^{-5} s^{-1} were used for this setup. The computational mesh is based on the mesh with 3397 elements used in [41], postprocessed using uniform refinement via edge bisection to a mesh with approximately 54,400 elements. The time step is set to 2 s for P0, 1 s for P1, and 0.5 s for P2.



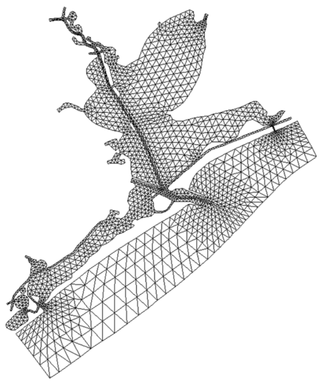


Fig. 2 Galveston Bay (from [41]): Bathymetry and boundary type (left), original triangular mesh with 3397 elements (right)

4 Parallelization with SYCL

SYCL is an open standard developed by the Khronos Group to target heterogeneous hardware from a single-source code. Following SYCL 1.2.1, the current standard is denoted as SYCL 2020. There are currently two major SYCL compilers supporting the SYCL 2020 standard: Intel oneAPI and AdaptiveCpp [11, 12]. Both compilers support GPUs from Nvidia, AMD, and Intel, although oneAPI requires additional plugins developed by Codeplay for Nvidia and AMD GPUs. These plugins are available from version 2023.0. Additionally, Intel FPGAs are supported through oneAPI if the Intel Quartus Prime software (required for synthesis) is installed.

In this section, we first present the general parallelism concepts of the implementation with work item parallelism, fine grained data parallelism, and double buffering between sequential iterations. We then look into memory optimizations regarding a portable layout and reducing bandwidth demand by re-calculation of values, before looking into FPGA-specific memory subsystems (Sects. 4.2.2 and 4.2.3) which get shaped from the SYCL code via the synthesis tools.

4.1 Algorithm implementation in SYCL

The Algorithm 1 described in Sect. 3.2 has ample potential for parallelism in its loops over elements and edges. The discretization and time-stepping scheme described by equations (8) and (12) can be implemented element-wise. The edge integrals need the projected solutions from the previous Runge–Kutta stage, and the projected solutions need to be updated after the new solution has been computed. By employing a double-buffering strategy for the projected solutions and initializing them before the first kernel invocation, every step can be calculated within a single kernel without race conditions, resulting in the design outlined in Algorithm 2.

Algorithm 2 Projection algorithm of UTBEST with merged projection and integration kernels

```
1: while $t < t_1$ do
2:     Loop over Runge–Kutta stages:
3:     for stages of the Runge–Kutta method do
4:         Element loop:
5:         for element indices $e \in \{1, \dots, E\}$ do
6:             calculate element integrals for terms two, four in Eq. 8
7:             calculate edge integrals for term three in Eq. 8, update only own element
8:             calculate $\boldsymbol{c}_\triangle$ for the next Runge-Kutta stage
9:             perform minimum depth control on $\boldsymbol{c}_\triangle$
10:            for edges belonging to $\Omega_e$ do
11:                $\boldsymbol{p}_\triangle \leftarrow$ projection of $\boldsymbol{c}_\triangle$ to edge
12:                store $\boldsymbol{p}_\triangle$ in global array
13:            end for
14:        end for
15:        synchronize, swap buffers for projections
16:    end for
17:    $t \leftarrow t + \Delta t$
18: end while
```

The element loop (Algorithm 2, line 5) can now be executed with any form of parallelism that fits the target architecture. The SYCL implementation encapsulates the loop body (Algorithm 2, lines 6–13) into a function denoted as `process_element`, which, including sub-functions, contains around 500 lines of numerical code and is shared for all target architectures. However, the simple kernels that invoke this `process_element` function differ in order to be able accommodate the fundamentally different execution models of CPUs and GPUs on the one hand (these can be expressed with work-item parallelism) and of FPGAs on the other (work best with pipeline parallelism for this kind of application). For CPUs and GPUs, the element loop is mapped to a `parallel_for` kernel in SYCL. Each work-item of this kernel corresponds to one element in the mesh, and the kernel is launched once per Runge–Kutta stage to achieve the global synchronization after each Runge–Kutta update. SYCL provides two different variants of a `parallel_for` kernel: The basic `range` variant, and an `nd_range` variant which also provides control over work groups. While we do not need the work group mechanisms from SYCL, we chose to use the `nd_range` variant with a group size of 256 elements. Using a fixed work group size leads in some cases to better performance, as for example also shown in [7], although the optimal work group size depends on the mesh size, target hardware, and most likely also on the SYCL implementation.

In contrast, for FPGA designs, the element loop is wrapped into a `single_task` kernel in SYCL, which gets translated into a pipelined design that starts to process exactly one new element in every clock cycle. In order to reduce kernel launch overheads, we additionally offload the entire time stepping code (Algorithm 2, lines 1–3, 15–18) onto the FPGA, whereby the dependency analysis of the FPGA compiler ensures a correct execution schedule of the design. Simplified code listings for the invocation code can be found in Appendix A.

Another small customization of the two code paths takes place within the `process_element` code. The FPGA design requires all of the small, data parallel loops (e.g. over quadrature points and degrees of freedom) within this function to be fully unrolled, which widens the compute pipeline at these locations and is prerequisite to avoid bottlenecks. For the CPU and GPU designs, after empirical evaluation, it is only beneficial to unroll 18 out of 38 such loops. A similar effect was noted by the authors of [47], where loop unrolling significantly improved performance, although to a much greater extent than in our code. A macro is used to distinguish the two cases. Further optimization and customization concerns the memory access as discussed next.

4.2 Memory access

While SYCL allows to write kernel code in C++ and removes the burden of targeting different programming models from the developers, it does not provide abstractions for different memory access patterns. Initially, this increases the development effort required for a portable implementation, but it enables us to fully customize memory accesses. Ultimately, we want to separate the specific memory layout from the numerical algorithm. In the following sections we describe the different data layouts required for different architectures.

4.2.1 Portable memory layout

In the initial UTBEST implementation, data is stored in an array of structs layout. This is still fitting for the FPGA design, where the unrolled inner loops mostly lead to concurrent access of complete structs per element (depicted in the left part of Fig. 3). For GPUs in contrast, best memory performance is achieved when threads with consecutive IDs load data from the same cache line, i.e., if data are stored in a struct of arrays layout (c.f. right part of Fig. 3). Experiments show that this layout is also beneficial to the CPU designs, particularly when the compilation flow translates work-item parallelism over elements into vectorized instructions (see results for oneAPI and AdaptiveCpp with OpenCL backend in Sects. 5.3 and 5.3.2 and Fig. 7).

To accommodate both memory layouts in the same code base, we implemented data structures for an array of structs and a struct of arrays memory layout as well as light-weight accessor classes for these data structures. In the numerical algorithm all data accesses are handled through these accessor classes, effectively separating the algorithm from the underlying memory layout. A similar strategy is, for example, also used by Kokkos [5].

Another challenge, independent of the target architecture, is the unstructured memory access pattern caused by the mesh. As the need for this is inherent to the problem, it cannot be avoided entirely. In the earlier implementation presented in [13], unstructured memory accesses occur during the edge integral computation for two reasons. First, to load edge-specific information like the normal vector and, second, to load the

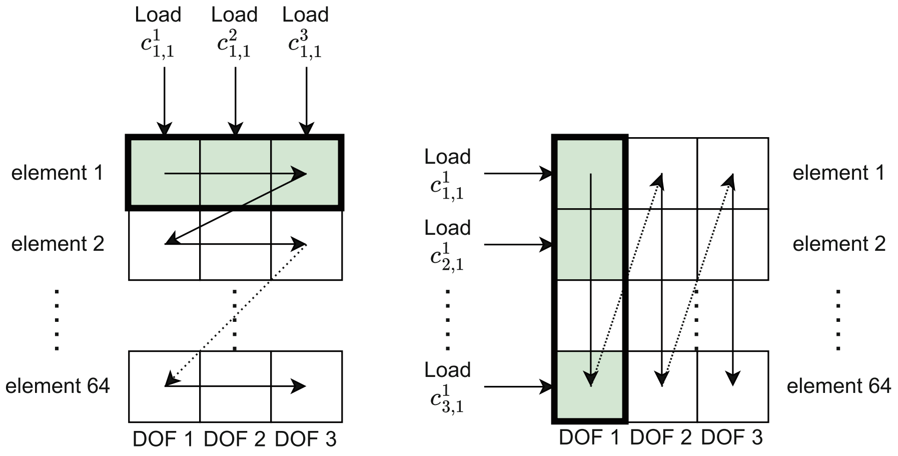


Fig. 3 Memory layout for the solution vectors (c^j_{ei} in Eq. (11) in the original array of structs implementation now used for FPGA targets (left) and for the struct of arrays alternative initially developed for GPU targets and also found to be preferable for CPU targets (right). Arrows indicate consecutive elements in the array. The highlighted boxes form ideally a single memory interaction from the different load instructions

projected solutions from the neighboring element. To improve the first type of loads, we now duplicate that edge information to keep one copy each along with each element. This duplication slightly increases the required memory capacity and the data volume to be loaded during the computation (compared to cache use in the original implementation with unique edge information), but it replaces unstructured memory access with more efficient linear access. In addition, this change does not affect stores, since this information remains constant during the simulation – in contrast to the projected solutions, which are therefore not duplicated.

Additionally, by having the edge information available as part of the element information, we are able to partially offset the additional memory overhead by computing the bathymetry at the quadrature points on demand rather than storing them as part of the mesh information. Generally, this approach improves the performance over the full range of target architectures. However, there is a corner case when we reconstruct the projected solutions for the edge integrals. The degrees of freedom for the local element have already been loaded, and reconstructing the projected solutions for the current element removes a global memory load from the edge integration. Projected solutions for the neighboring element still have to be loaded, causing an unstructured global load. Reconstructing the local projection values on demand is beneficial for GPU and FPGA performance, but reduces performance for CPU execution. Specialized and architecture-dependent code paths for this specific change would reside within the numerical code, in contrast to the other target-specific variants discussed. In the interest of a portable code base, we decided to use a unified variant with the projection reconstruction always enabled throughout the remainder of this work. We quantify the involved trade-off in Sect. 5.3.3.

4.2.2 Array-specific caches for the FPGA design

Unlike CPUs and GPUs, FPGAs do not have a memory hierarchy with caches, in the case of GPUs alongside local memory per Streaming Multiprocessor, that absorb some of the latency when repeatedly accessing the same data from the main memory. This means that data needs to be stored either in main memory, which is accessed slowly, or in explicitly instantiated on-chip buffers using dedicated hardware resources. In the OpenCL designs presented in [23, 37], all data is stored in buffers synthesized in on-chip FPGA block RAM. Because these designs do not have global memory accesses, they perform very well but are limited to small grids.

In order to not limit the generality of the new SYCL code base but still profit from local memory where possible, we implemented a caching approach for most of the global memory buffers (cf. left part of Fig. 4). All data except the parameters for the boundary forcing (Eq. 13) are cached. The forcing parameters are only relevant for a small number of elements, and experiments showed that there is no significant influence on the performance if these values are loaded from the off-chip memory. The caches are simple in that they map a fixed subsection of the buffer into an on-chip memory copy. However, they can be quite sophisticated in their layout and interface, such that for example 6×3 concurrent loads or stores from an unrolled loop nest can be served in the same clock cycle via independent local memory banks.

The read-only caches are filled once at kernel launch. In contrast to a dynamic cache management, this yields the benefit of reducing the amount of random accesses to the global memory, since the caches are filled linearly, and only reads from the cache follow the random access pattern. For smaller grids that fit completely into the cache, it completely avoids slow main memory accesses during execution, just like in the earlier OpenCL designs. For larger grids, only the first elements come from the cache and the others are loaded from the main memory, a scenario where dynamically managed caches may achieve higher hit-rates depending on the temporal and spatial locality of access patterns. For arrays that are not only read but also modified, the cache contents are written back to main memory at the end of the kernel launch. All caches are implemented as C++ classes separated from the numeric code so that they can be used transparently for the FPGA target.

4.2.3 Memory interface variant for FPGAs with HBM

As an alternative to the array-specific caches, described in Sect. 4.2.2, we also explore the usage of FPGAs with HBM memory to mitigate the memory bandwidth limitation without resorting to our custom-designed caches. For this purpose, a special variant of the SYCL design was built. To understand its implementation rationale, some background on the FPGAs' off-chip memory is required.

While the DDR off-chip memory offers a small number of memory channels – a Bittware 520N card has four DDR4 memory channels and other cards just provide one or two – HBM typically offers a higher number of memory channels: a Bittware 520N-MX card provides 32 HBM-Memory channels for a much higher aggregate bandwidth (c.f. right part of Fig. 4). However, there are more constraints to use such HBM interfaces efficiently. With a board support package (BSP) for the DDR memory interface, each load-store unit (LSU), the hardware unit generated for each buffer access, can access data from all DDR channels, and an interleaved interface mode can automatically distribute buffers to all four channels. In contrast, with HBM memory, each buffer needs to be mapped to one specific HBM channel and each LSU should be connected exactly to one single HBM channel. With the current oneAPI tools and FPGA platform, this mapping needs to be specified as part of the SYCL implementation – this might change with next generation FPGAs with hardened NoCs providing more flexible routing to the individual HBM channel endpoints.

Furthermore, for full pipeline throughput, all loads and stores must be within the width of a single memory channel. For example, the data structure containing the information about the element is 1024 bits large, and each memory channel is 256 bits wide. By splitting the data structure into four sub-parts, each of them in an individual buffer, a single element can be loaded in a single clock cycle if the buffers with the sub-parts are distributed over four different memory channels. This approach essentially mimics the automatic interleaving from the DDR memory interface – however with precise adoption of the involved number of channels to the required read width.

A similar strategy can be applied when multiple independent accesses into the same buffer are required concurrently, which is the case for the projection values for all three edges per element. By replicating the buffer and assigning each copy to its own memory bank, these loads can be performed in parallel – in conjunction with

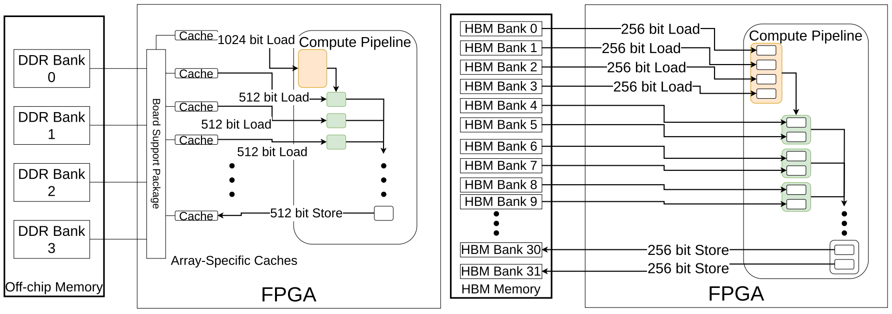


Fig. 4 Memory access layout for the different FPGA Designs. The schematic on the left shows the cached variant with buffers in on-chip memory. On the right the HBM variant is shown, where each load or store is assigned to a specific HBM channel

interleaving as discussed before and adapted to the required number of bits depending on the polynomial order. When these buffers are modified, all copies of the buffer need to be updated as well. To achieve the full throughput for double-buffered arrays, the source and the destination buffers must have their own memory banks so that the reads and writes can happen concurrently.

The goal of applying this to all memory access is to make it possible to load all the needed data for an element update concurrently within a cycle. In theory, it is then possible to compute an element update in one clock cycle, similar to on-chip memory designs. However, random accesses are expected to introduce some overhead in the memory subsystem so that the full throughput is not achieved in practice. Similarly to the caches in Sect. 4.2.2, the memory access pattern spread over different HBM memory channels is implemented in C++ classes, and the numerical code remains the same as for the other architectures. In contrast to other variants, this design needs a custom memory allocation to handle and create all the buffer replications.

5 Performance results and analysis

For all performance results we use, unless stated otherwise, oneAPI 2023.2 on CPUs and GPUs with the respective plugins by Codeplay and AdaptiveCpp v24.02 with LLVM 16. The FPGA designs were synthesized with oneAPI 2024.0. All binaries were compiled with `-O3 -ffast-math -march=native` flags. Additionally, we pass `-fp-model=precise` to the oneAPI compiler to avoid the use of lower-precision math functions on CPUs.

5.1 Validation of the SYCL implementation

Our SYCL implementation realizes essentially the same discretization as the original UTBEST [41] code; the only difference in the discrete scheme is the use of edge projections (see line 11 of the Algorithm 2) instead of the direct evaluation of the element solutions at the edge quadrature points. This particular modification as well

Table 1 Deviation from the UTBEST results for the water elevation $\Delta\xi$ and velocity magnitude $|\Delta\boldsymbol{q}|$ of the SYCL implementation with oneAPI and AdaptiveCpp (ACPP)

Compiler	Variable	P0	P1	P2
oneAPI, Intel Xeon 8358 `-ffast-math`	$\Delta\xi$, m	$1.3 \cdot 10^{-5}$	$1.3 \cdot 10^{-5}$	$1.3 \cdot 10^{-5}$
	$\lvert\Delta\boldsymbol{q}\rvert$, m^2/s	$7.2 \cdot 10^{-5}$	$8.5 \cdot 10^{-5}$	$1.3 \cdot 10^{-4}$
oneAPI, Intel Xeon 8358 `-fp-model=precise`	$\Delta\xi$, m	$1.0 \cdot 10^{-7}$	$3.7 \cdot 10^{-7}$	$5.4 \cdot 10^{-6}$
	$\lvert\Delta\boldsymbol{q}\rvert$, m^2/s	$1.1 \cdot 10^{-6}$	$3.9 \cdot 10^{-6}$	$8.7 \cdot 10^{-5}$
oneAPI, AMD MI210	$\Delta\xi$, m	$2.0 \cdot 10^{-7}$	$3.4 \cdot 10^{-7}$	$1.2 \cdot 10^{-5}$
	$\lvert\Delta\boldsymbol{q}\rvert$, m^2/s	$1.1 \cdot 10^{-6}$	$1.5 \cdot 10^{-5}$	$6.5 \cdot 10^{-5}$
oneAPI, Stratix 10 GX 2800	$\Delta\xi$, m	$1.0 \cdot 10^{-7}$	$3.4 \cdot 10^{-7}$	$1.5 \cdot 10^{-5}$
	$\lvert\Delta\boldsymbol{q}\rvert$, m^2/s	$1.1 \cdot 10^{-6}$	$8.6 \cdot 10^{-6}$	$6.1 \cdot 10^{-5}$
ACPP, Intel Xeon 8358 `-ffast-math`	$\Delta\xi$, m	$1.0 \cdot 10^{-7}$	$3.3 \cdot 10^{-7}$	$1.2 \cdot 10^{-5}$
	$\lvert\Delta\boldsymbol{q}\rvert$, m^2/s	$1.1 \cdot 10^{-6}$	$8.9 \cdot 10^{-6}$	$4.5 \cdot 10^{-5}$
ACPP, AMD MI210	$\Delta\xi$, m	$1.0 \cdot 10^{-7}$	$4.1 \cdot 10^{-7}$	$8.4 \cdot 10^{-6}$
	$\lvert\Delta\boldsymbol{q}\rvert$, m^2/s	$1.0 \cdot 10^{-6}$	$9.4 \cdot 10^{-6}$	$6.5 \cdot 10^{-5}$

as purely algorithmic adaptations shown in Algorithm 2 can only affect the round-off errors of the numerical scheme.

To verify this, Table 1 lists the maximum deviation in the water elevation $\Delta\xi$ and velocity magnitude $|\Delta\boldsymbol{q}|$ from the corresponding UTBEST results on the Bahamas mesh with 10,240 elements after a one day simulation with a time step of one second. The typical values for the water height are around 0.2 meters and, for the depth-integrated velocity, up to 1.4 m^2/s. The deviations are small for all approximation orders, and the differences between platforms can be attributed to the round-off effects (all implementations use the single-precision floating-point arithmetic) and different parallelization strategies. Slightly larger deviations for piecewise quadratic approximations are due to much longer polynomial solution representations and more quadrature points used for the P2 discretization. The CPU and GPU values in Table 1 were measured on the Intel Xeon Platinum 8358 CPU and AMD MI210 GPU; for other CPUs and GPUs values are slightly different but generally of the same magnitude.

Without passing `-fp-model=precise` to the oneAPI compiler, lower-precision math functions are linked into the binaries which cause slightly larger deviations, as indicated in the first row of the table. We measured a performance decrease of at most five percent with `-fp-model=precise`, and in some cases the produced code was even faster compared to the version with lower precision math functions.

5.2 Performance comparison between architectures

All performance data for the CPUs and GPUs were collected on the test cluster at the Erlangen National High Performance Computing Center. On CPUs, turbo boost has been disabled, and the clock frequency was fixed to the nominal frequency as given by the hardware vendors. The FPGA results were obtained on the Noctua 2 [48] system at the Paderborn Center for Parallel Computing. The FPGAs run on a fixed clock frequency mandated by the synthesized design.

Because of the huge difference in performance between the different architectures, we use a different number of time steps depending on the mesh size and whether we run on CPUs, GPUs, or FPGAs. In general, we aim for a total execution time of the simulation loop between 10 s and 15 min. Preparatory experiments showed that the results are fairly reproducible with this setup. FPGAs show a deviation of less than one percent, CPUs and GPUs have in general higher deviations with results from different runs being within 5 – 10 % of each other.

Figure 5 illustrates performance of our code by plotting the throughput in degrees of freedom per second (DOF/s) for different approximation orders, mesh sizes, and architectures. With increasing polynomial order from piecewise constant to piecewise linear, and further to piecewise quadratic, the number of calculated DOFs increases faster than the involved data volume, thus generally allowing for higher throughput metrics, particularly in bandwidth limited regimes. The tested architectures are: Nvidia H100 GPU, as contained within a Grace-Hopper module, AMD MI210 GPU, Intel Xeon Platinum 8470 (code name "Sapphire Rapid") CPU, AMD EPYC 9684X ("Genoa-X") CPU, Nvidia Grace CPU, and two Intel Stratix 10 FPGA variants. Hardware parameters (number of cores, memory bandwidth and peak GFLOPS) of the used CPUs and GPUs are listed in Appendix B, while a summary of hardware data for the FPGAs can be found in Table 4.

As indicated by the solid lines, the GPUs perform best for most data points, but need relatively large mesh sizes before they reach their peak performance. With CPUs we see a qualitatively similar behavior, however with a lower peak performance partially related to lower platform bandwidth and lower architectural peak FLOPS. It is not surprising that both GPUs and CPUs need relatively large meshes to reach their peak application performance, as the elements are grouped in blocks that are distributed across cores, which in turn also each need to process many elements for good utilization – on GPUs mainly for memory latency hiding, on CPUs for vectorization and dynamic instruction reordering within and beyond loop iterations. This results in underutilized resources for the smallest meshes and consequently lower performance. The performance of the FPGA version with HBM is essentially independent of the mesh size, and the cached FPGA implementation reaches its peak performance for meshes within its cache capacity and shows overall best performance for the smallest meshes.

5.3 Evaluation of CPU performance

AdaptiveCpp and Intel oneAPI have different vectorization strategies on CPUs. oneAPI utilizes the Intel OpenCL runtime to execute SYCL code on the CPU. The OpenCL runtime vectorizes the kernel across work-items (in our code: elements) to utilize SIMD instructions on CPUs. AdaptiveCpp by default utilizes OpenMP for CPU execution and consequently relies on the auto-vectorization capabilities of the Clang

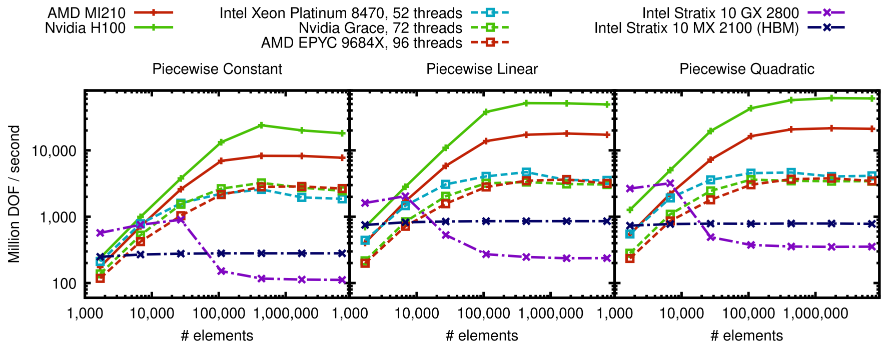


Fig. 5 Performance measured in million degrees of freedom per second. Solid lines indicate GPUs, dashed lines CPUs (single-socket, one thread per core). The dash-dotted lines indicate the Intel FPGAs: The Stratix 10 GX in purple and the Stratix 10 MX with HBM in dark blue. On CPUs and GPUs, the code has been compiled with AdaptiveCpp 24.02, for the FPGAs we use Intel oneAPI 2024.0. OpenCL with vectorization width 16 is used for parallelization on the Intel CPU, while the AMD and Nvidia CPU execute OpenMP code (colour figure online)

compiler. The compiler then vectorizes the loops over the 2D quadrature points rather than the elements. OpenMP can either be targeted through an ahead-of-time compilation flow or through the just-in-time CPU compiler introduced in version 2024.02. Furthermore, we can also target Intels OpenCL runtime through AdaptiveCpp. In that case, we have to explicitly control the vectorization width in the OpenCL runtime (which is possible through an environment variable).

5.3.1 Strong scaling

For a first analysis, we perform a strong scaling experiment for a mesh with 7 million elements on three different CPUs: An AMD EPYC 9684X (codename "Genoa-X", 96 cores at 2.55 GHz), an Nvidia Grace CPU (ARM based, 72 cores at 3 GHz), and an Intel Xeon Platinum 8470 ("Sapphire Rapid", 52 cores at 2 GHz). All tested systems have two sockets, we use one thread per physical core resulting in scaling tests with up to 192 threads. For the scaling runs, we use the AdaptiveCpp compiler for the Nvidia Grace and AMD EPYC CPUs; on the Intel system, we use oneAPI as it yields higher performance compared to AdaptiveCpp. Intel does not provide options to pin threads to cores (and tools like likwid-pin do not work either), so we just restrict the available CPU cores through the "taskset" command. With OpenMP, we can explicitly pin threads by setting `OMP_PROC_BIND=close` and `OMP_PLACES=cores`; the number of threads is again controlled by the "taskset" command. Only the time for the simulation loop is measured and compared, I/O operations are not included and outputs between time steps have been switched off.

Figure 6 shows the strong scaling efficiency on the tested systems. In general, the higher discretization orders scale better than the piecewise constant approximation, and AdaptiveCpp's OpenMP code on AMD and Nvidia CPUs shows better scaling behavior than the oneAPI code on the Intel CPU. For the P1 and P2 discretizations,

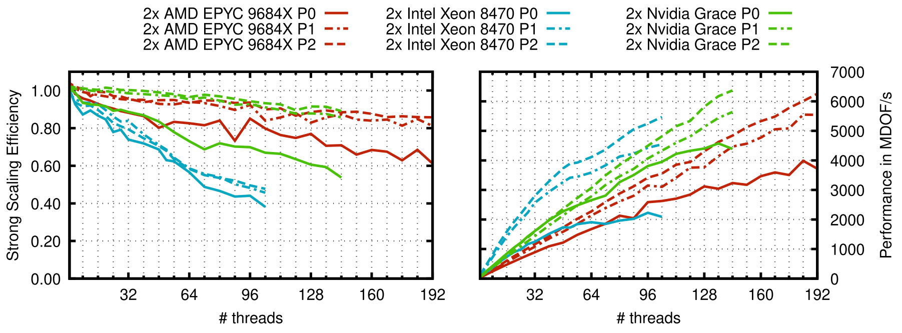


Fig. 6 Strong scaling efficiency and absolute performance on dual-socket AMD, Intel and Nvidia CPUs (one thread per core) on a mesh with 7 million elements. AMD EPYC and Nvidia Grace results with AdaptiveCpp 24.02 and OpenMP backend, Intel Xeon results with oneAPI 2023.2 and OpenCL backend

single-core performance is highest on the Intel Xeon CPU at 95.6 and 110.3 MDOF/s respectively and maintains the performance lead at identical core counts up to 104 cores. On a full dual-socket system, Nvidia's Grace and AMD's Genoa-X CPU both outperform the Intel system due to the better scaling behavior of the OpenMP code and the higher core count. While the scaling efficiency on the AMD and Nvidia CPU is similar for P1 and P2, a lower single-core performance on the AMD CPU results in similar performance on the full system run despite the higher core count.

Due to a much lower arithmetic intensity, scaing for the P0 discretization is worse on all platforms, with parallel efficiency on the Intel CPU dropping to around 0.4 at full core count. In terms of absolute performance, the Nvidia CPU, starting on a single core with 56.6 MDOF/s, maintains the performance lead up to the full system run.

Whereas for P1 and P2, performance on the Nvidia and AMD CPUs does not seem to approach any platform limitations like memory or communication bandwidth yet, for P0, and for the Intel CPU possibly also for the other discretizations, performance saturation due to such effects may be approached. Overall, it seems that the performance obtained with the code reflects the different strengths of the target architectures well.

5.3.2 Vectorization with different compilers

The next comparison focuses on the Intel and AMD CPUs, for which different compiler and backend options exist, whereas for the ARM-based Nvidia Grace CPU compatibility limitations exist both for oneAPI as well as for AdaptiveCpp with PoCL as the OpenCL backend. For the strong scaling tests, we use Intel oneAPI on the Intel CPU as it yields higher performance, while AdaptiveCpp performs better on the AMD CPU. To better understand this behavior, we again run a strong scaling test (this time with a mesh with only 1.7 million elements) and compare performance between oneAPI and AdaptiveCpp with different backends. We set Intel oneAPI as the baseline and compute the relative performance for different number of threads. As backends for AdaptiveCpp we tested OpenMP in ahead-of-time compilation mode as well as Intel's

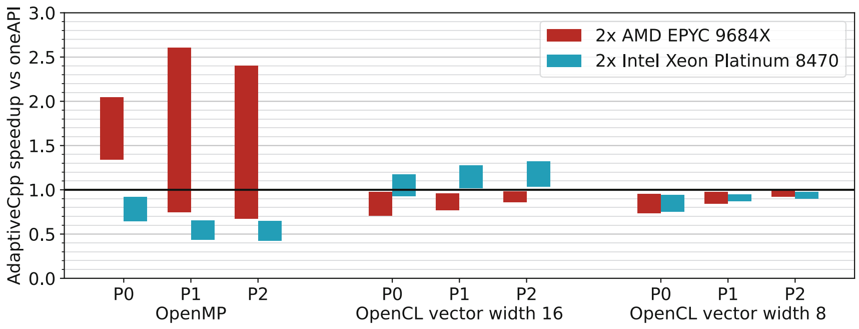


Fig. 7 AdaptiveCpp performance relative to oneAPI. Bars indicate best and worst relative values over all core counts (up to full dual-socket scale). On the AMD CPU, AdaptiveCpp's OpenMP code performs better than oneAPI's OpenCL code; on the Intel CPU, OpenCL yields better performance than OpenMP, and AdaptiveCpp has the highest speedups over oneAPI for low core counts.

OpenCL runtime with vectorization width 8 and 16 (corresponding to AVX and AVX-512 instructions respectively). AdaptiveCpp's just-in-time compilation flow for CPUs generates OpenMP code at runtime. In our experiments, this results in nearly identical performance on the tested systems and is therefore not evaluated further. Figure 7 shows the aggregated numbers as minimum and maximum speedup over all tested core counts of AdaptiveCpp with different backends over the oneAPI baseline. Note that the experiments use the struct of arrays layout as presented in Sect. 4.2.1 and the right subplot of Fig. 3.

On the AMD CPU (red bars), AdaptiveCpp with OpenMP ahead-of-time compilation yields the best overall performance. With approximation orders P1 and P2, oneAPI is faster until 48 and 64 cores, respectively, after which the OpenMP code with its better scaling behavior outperformed the OpenCL backend. By targeting the Intel OpenCL runtime via AdaptiveCpp, we see performance more similar to oneAPI.

On the system with the Intel Xeon CPU (blue bars), oneAPI outperforms AdaptiveCpp with OpenMP for all approximation orders and up to the full dual-socket run. Contrary to the AMD system, vectorization width 16 performs better than vector width 8, and AdaptiveCpp then yields up to 30 % higher performance than oneAPI for low core counts. When using the full system, the measured performance is nearly identical between both SYCL implementations (lower end of the blue bars for OpenCL vector width 16).

Generally, the performance difference between AdaptiveCpp with OpenMP and both compilers with OpenCL backend originates from different strategies for generating vector instructions. AdaptiveCpp vectorizes over small loops within the `process_element` function (see Sec. 4.1), e.g., over degrees of freedom or quadrature points. This is good for data reuse in registers and caches, but, given the small non-power-of-two iteration counts, like 3, 9, or 18, often leaves some vector lanes unused. Also frequent changes between scalar and vector operations are required. In contrast, the OpenCL backend generates vector instructions over multiple different elements on the level of the outer `parallel_for`, which allows for efficient use of all 8 or 16 vector lanes, but is more bandwidth intensive, since much less data per element can be kept in registers when processing multiple elements concurrently. Overall, the experiment shows that regardless of the code portability itself, the combination of target CPU, compiler, and CPU backend has performance implications that may require manual comparisons or auto-tuning for best performance.

5.3.3 Performance trade-offs for portability

Beyond the performance implications of different backends and CPU architectures, aiming for a shared implementation of the numerical algorithm can involve trade-offs between optimal performance for a specific hardware and achieving good (or at least reasonably good) performance across a range of target architectures. As discussed in Sect. 4.2.1, one notable case arises during the calculation of edge integrals. To compute the flux across the edge, the local solution has to be evaluated for each quadrature point on the edge. These evaluations are done at the end of each time step in a projection phase, for the results to be used in the next time step. During the integration, in the first SYCL version presented in [13] and going back to the first FPGA design in [23], these projected solutions were loaded both for the local and neighbor element. Now, instead of loading the projected values for the local element, the code re-computes these values during the edge integration, trading reduced bandwidth demands for more local computations.

Table 2 shows the performance implications of this change for selected target architectures, with the relevant trade-off showing up for CPU targets. On CPUs with OpenMP backend, which is required for the Nvidia CPU and is superior for the AMD CPU, performance dropped by as much as 20 %. For CPU execution with the OpenCL backend, which is superior for the Intel CPU, performance remained constant or increased a bit. The different trends for OpenMP and OpenCL backends can be attributed to the different vectorization strategies. With the less efficient vectorization, OpenMP is more instruction bound, such that the extra operations for projection reconstruction are a clear overhead. In contrast, with the OpenCL backend more bandwidth bound, the extra operations are compensated by bandwidth savings.

For GPUs, the projection reconstruction is beneficial, increasing performance on the AMD MI210 and Nvidia H100 GPUs by up to 10 %. The new approach also freed resources on the FPGAs, enabling an increased cache size for the P2 discretization on the Stratix 10 GX card and allowing for a good utilization of HBM2 channels on the Stratix 10 MX card (see Sect. 5.5 for details).

5.4 Evaluation of GPU performance

The GPU results in Fig. 5 were obtained with AdaptiveCpp, but in many cases oneAPI achieved very similar results. However, analysis of underlying performance metrics reveals some interesting differences between the two compilers. We exemplarily highlight these differences on an AMD MI210 GPU in Table 3.

While little difference is evident between AdaptiveCpp and oneAPI when comparing only the wall-clock time for the P0 and P1 discretizations, a closer look reveals

Table 2 Absolute performance values on selected CPUs and GPUs, where projected edge values are either loaded from memory or computed on-demand. Performance was measured on the largest mesh with 7 million elements

Device	New approach			Old approach		
	Projection reconstruction			Projection load		
	MDOF/s			MDOF/s		
	P0	P1	P2	P0	P1	P2
2× Nvidia Grace CPU, OpenMP	**4389**	**5638**	**6371**	4037	5786	7024
2× AMD EPYC 9684X CPU, OpenMP	**3774**	**5865**	**6260**	4402	6877	7789
2× AMD EPYC 9684X CPU, OpenCL	1622	3267	4007	1637	2855	3642
2× Intel Xeon 8470 CPU, OpenMP	2246	3640	4375	2310	3769	4764
2× Intel Xeon 8470 CPU, OpenCL	**2178**	**4170**	**5353**	2126	4068	5090
AMD MI210 GPU	**7746**	**17,211**	**21,201**	6667	15,476	18,476
Nvidia H100 GPU	**18,152**	**49,345**	**60,714**	18,209	46,850	58,027
Mesh with 6784 elements						
Intel Stratix 10 GX 2800	**753**	**2039**	**3210**	716	1940	644
Intel Stratix 10 MX 2100	**268**	**823**	**776**	202	602	618

For the FPGA, we illustrate the results on the mesh with 6784 elements as it shows the most relevant difference. Bold font denotes configurations used in the other parts of the article

some significant differences how this performance is achieved by either compiler. We use AMDs "rocprof" tool to gather performance counters on the GPU, in particular for the floating-point instructions (addition, multiplication, fused multiply-add, and transcendental functions) and bytes read and written. These data are then averaged over the number of elements in the mesh and is summarized in Table 3.

AdaptiveCpp's code executes between 30% and 35% more floating-point operations per element for the same discretization, and also accesses 6% – 15% more data from the device memory. Yet, it achieves slightly lower kernel execution times than oneAPI. For the P2 discretization, we measured a significantly higher data volume with oneAPI: more than twice as much data is loaded from the main memory. Further inspections reveal that this phenomenon is caused by register spilling: The oneAPI code requires many more registers and has to spill registers into scratchpad memory, which is backed by global memory. Upgrading the oneAPI compiler to version 2024.1 resulted in even more register spilling and consequently lower performance.

Register spilling on GPUs is a known performance problem, for example also discussed in [9] for Intel GPUs. There register spilling is controlled by increasing the maximum number of registers per work group and controlling the sub-group size, but also note that the optimal combination of these parameters depends on the kernel. In

Table 3 Performance metrics on the AMD MI210 GPU: FLOPs/element, Bytes/element and average kernel time for AdaptiveCpp and oneAPI

	AdaptiveCpp			oneAPI		
	P0	P1	P2	P0	P1	P2
Kernel time (μs)	1987	4048	5715	2248	4272	14,720
FLOPs/element	356	1266	2944	273	936	2195
bytes/element	263	436	704	247	378	1734
GFLOPS	1168	2173	3579	845	1522	1036
GB/s	920	749	855	762	615	818

Values are rounded to the nearest integer

our case, the AMD MI210 requires a sub-group size of 64 elements (the wavefront size), and we found no compiler options to improve register usage.

It is currently not clear where the huge difference in floating-point operations originates from. A simple model that counts operations within the code estimates around 400 FLOPs/element for P0, 1350 FLOPs/element for P1 and 3800 FLOPs/element for P2. While this model potentially overestimates the actual number of floating-point operations by not accounting for all compiler optimizations, a difference of up to 40% is still somewhat surprising.

5.5 Performance aspects on FPGAs

In contrast to the CPU and GPU targets, the FPGA kernels are not just compiled, but a high level synthesis (HLS) process translates them into a circuit description that then gets mapped to logic, DSP, and block RAM resources. Table 4 shows the results of the entire synthesis process. The HLS step aims for a target clock frequency (default 480 MHz for P0 and P1, manually lowered to 240 MHz for P2 with caches) that gets automatically decreased according to the critical path in the design mapped to actual hardware resources. The pipelined execution model of the FPGA design along with fine-grained data level parallelism leads to designs that can ideally process one entire element per clock cycle, i.e., 3, 9 or 18 DOF/cycle for P0, P1, and P2, respectively. The increase in work performed per clock cycle is mostly visible in the DSP usage in Table 4, since floating-point operations are mostly mapped to DSPs. There are two deviations from the simple execution model of one element/cycle. One deviation is the impact of pipeline latency, which occurs once per time step or RK step and was analyzed in more detail in [23]. The other big caveat is data availability. When off-chip bandwidth is not sufficient to feed the computation pipeline, it stalls, similarly to CPU or GPU execution.

Table 4 Summary of the resource usage for the FPGA designs in relation to the total available resources on the device

order	Cache size	RAM	DSP	Logic	Clock (MHz)
Cache designs on Intel Stratix 10 GX 2800					
P0	32,768	6,517 (56 %)	593 (10 %)	264,025 (28 %)	322.50
P1	16,384	6,461 (55 %)	1,243 (22 %)	307,786 (33 %)	268.00
P2	8,192	5,587 (48 %)	2,514 (44 %)	345,066 (37 %)	211.67
HBM Designs on Intel Stratix 10 MX 2100					
P0	–	2,204 (32 %)	593 (15 %)	334,260 (48 %)	232.00
P1	–	2,697 (39 %)	1,237 (31 %)	365,161 (52 %)	232.50
P2	–	3,122 (46 %)	2,520 (64 %)	434,531 (62 %)	197.62

In total, the Intel Stratix 10 GX 2800 contains 11,721 RAM Blocks, 5,760 DSP blocks, and 933,120 logic blocks. In total, the Intel Stratix 10 MX 2100 contains 6,847 RAM Blocks, 3,960 DSP blocks, and 702,720 logic blocks. The cache size means the number of elements for which the data can be stored on the device, as described in Sect. 4.2.2

5.5.1 Cache designs

First we discuss the results for the designs using the caches described in Sect. 4.2.2 in more detail. These are running on the Intel Stratix 10 GX 2800 FPGA (on a Bittware 520N board), and the cache size is scaled to availability of RAMs. The results depicted in Fig. 5 show the combined effects of pipeline latency and of bandwidth limitations partially mitigated by caches in practice. Due to the reconstruction of the projection values (Sect. 4.2.1), some resources could be saved in the cache for this buffer since an internal copy for loading the local projection values is not needed anymore. This made it possible to increase the cache size for the P2 designs so that, for all polynomial orders and the two smallest meshes, the caches are large enough to hold all the data in the device's local memory, and no bandwidth-induced stalls occur. Compared to the designs with the load of the local projection values and the smaller cache size of 4,096, the performance increased from 644.72 MDOF/s to 3210.0 MDOF/s on the mesh with 6,784 elements (Table 2). For the smallest mesh with 1,696 elements, the FPGA designs with caches achieve the highest throughput for all the tested architectures, because the FPGA pipeline is already well utilized, but the other architectures lack data parallelism to reach their peak performance.

Looking further at Fig 5, for the mesh with 6,784 elements, the FPGA designs have higher throughput than for the smallest mesh. This can be explained by the pipeline latency of around 1,050 cycles that is now compensated by more elements processed with full throughput, because the cache capacity is still sufficient here. To analyze the combination of pipeline latency as well as filling and draining of caches, consider the P1 design when the pipeline is filled and all data comes from on-chip memory blocks. This design would produce a single element update per cycle which is equivalent to 9 DOF per cycle. With the frequency of the design of 268.00 MHz (c.f. Table 4), the maximum achievable throughput is 9 DOF $\times$ 268.00 MHz $=$ 2412.0 MDOF/s. For the mesh with 1,696 elements, this design achieves only 66% of this ideal throughput, and, for the mesh with 6,784 elements, this increases to 80%.

We see however that, for the mesh with 6,784 elements, the GPUs already overtake the FPGA in terms of performance or catch up to it. Then, for increasing mesh size, the impact of the caches decreases more and more: e.g., on the mesh with 27,136 elements only 60% of the data is fetched from the cache and the remainder must be loaded from global memory. For the largest meshes, the curves flatten out, and the throughput becomes closer to that of designs on the same card that do not use caches (not depicted in Fig. 5). The throughput without caches remains almost constant for all mesh sizes at around 100 MDOF/s for P0, 220 MDOF/s for P1 and 350 MDOF/s for P2. In comparison with other architectures this is partially caused by the lower peak global memory bandwidth (76 GB/s theoretical peak, 69 GB/s measured [49] with Stream Triad) but also due to suboptimal bandwidth efficiency without further customization of the design.

5.5.2 HBM designs

The HBM designs described in Sect. 4.2.3 run on the Intel Stratix 10 MX 2100 (on a Bittware 520N-MX board). In terms of DSP usage, their relative resource consumption (c.f. Table 4) is higher due to lower capacity of the target FPGA. In terms of logic resources, the absolute resource utilization is higher due to the additional memory interfaces. In contrast, the usage of on-chip RAM is lower due to the absence of caches.

Since all data are transferred via these LSUs from and to HBM memory without caches, the performance is almost constant over all mesh sizes at around 280 MDOF/s for P0 and 850 MDOF/s for P1 and P2 (cf. Figure 5) – i.e., around 2–3 times faster than the cached designs when they run in the bandwidth limited regime. Considering the effect of pipeline latency in the P1 design, we see only an increase of 100 MDOF/s from the smallest mesh to the mesh with 27,136 elements; in contrast to the other design, there is no overhead in filling caches at kernel launch. The P0 and P1 designs achieve a throughput of around 2.5 cycles per element.

$$\mathrm{P0}: \frac{232\,\mathrm{MHz}}{\left(\frac{280\,\mathrm{MDOF/s}}{3\,\mathrm{DOF/Elem.}}\right)} \approx 2.48\,\mathrm{Cycles/Elem.} \quad \mathrm{P1}: \frac{232.5\,\mathrm{MHz}}{\left(\frac{850\,\mathrm{MDOF/s}}{9\,\mathrm{DOF/Elem.}}\right)} \approx 2.46\,\mathrm{Cycles/Elem.}$$

The difference to the ideal throughput of one cycle per element can be explained for the most part by the overhead of the random access loads of the edge projection. Experiments with small test applications have shown this overhead for random access loads. Thus, the random access loads take around 2.5 cycles. This effect makes the LSUs and HBM channels used to load edge projection values a bottleneck of the design, leaving other HBM channels used for more regular memory accesses underutilized. In contrast, for the cached design on the Stratix 10 GX 2800 with DDR4 memory, all four physical memory channels are fully occupied with a mix of linear and random accesses. This is the main reason why the much higher bandwidth potential of the HBM equipped FPGAs is not translated into linear performance gains. If the random access overhead could be overcome by adding another layer of interleaving, namely of three subsequent loads for consecutive elements to three independent memory channels, such that asymptotically three elements could be processed in three cycles, the P1 performance could come out close to an idealized value of 232.5 MHz $\times$ 9 DOF $=$ 2,092 MDOF/s, roughly on par with the CPU results.

For the P2 design shown in Fig. 5, the throughput does not increase substantially compared to the P1 design since the random access loads contain 9 floats, which do not fit into the channel width of 256 bits. Therefore, the random access overhead is included twice, and the P2 design takes around 4.5 cycles per element.

As already mentioned in Sections 4.2.1 and 5.3.3, the reconstruction of the local projection values is beneficial for the HBM designs due to the omission of the load of the local projection values from the global memory. This allowed each buffer, especially the source and destination buffers in the double-buffering scheme, to be assigned to its own memory bank so that load and store operations could be performed simultaneously. In designs loading the local projection, there is an additional overhead of at least 1 cycle per element, so that for P1 they achieve around 600 MDoF/s and 3.5 cycles/element (cf. Table 2).

5.6 Evaluation of energy efficiency on different architectures

Energy efficiency is a rising concern in high-performance computing and specifically in climate simulations. As we are able to target a diverse set of hardware, we can naturally ask which architecture yields the best energy efficiency.

Through the use of LIKWID [50] we are able to measure energy consumption on recent Intel and AMD CPUs. While this is convenient for developers, one should keep in mind that AMD and Intel CPUs might measure different parts of the socket, and the measurements, at least for AMD Zen 2 ("Rome") CPUs, are known to be inaccurate [51]. Nvidia provides energy measurements for their new Grace CPU through the Linux Hardware Monitoring interface [52]. Nvidia's performance tuning guide also explains differences in power measurements between their Grace CPU and AMD Zen 4 ("Genoa") and Intel Sapphire Rapid CPUs: On the Grace CPU, power measurements are taken before the power regulators, whereas power sensors are placed after the power regulators on the Intel and AMD CPUs. According to Nvidia, regulator losses are approximately 15 % of the TDP power limit.

AMD and Nvidia also provide API calls to query the current power draw through their ROCm-SMI and NVML libraries respectively. The power consumption of the FPGA cards can be queried with command line tools provided by Bittware or via wrapper interfaces on Noctua 2 [48]. This requires continuous monitoring of the power draw during the execution of the code.

The Power Measurement Toolkit (PMT, [53]) is an open source library unifying different measurement APIs under one common interface. With this interface, we can decide at runtime which backend should be used for energy measurements. To be able to measure the Grace CPU independently from the GPU on a Grace-Hopper system, we added a separate backend for the Hardware Monitoring interface and also added the Noctua 2 specific code to query FPGA power draw.

At runtime, PMT creates a background worker thread which periodically reads hardware energy counters. These readings are then used to calculate a total energy consumption for the entire simulation loop. For the measuring setup, we only measure the accelerator device, i.e., only the CPU, GPU, or FPGA. The host CPU controlling the simulation is neglected, as the vast majority of computation time – and presumably also of the energy – is spent on the accelerator. Furthermore, we measure the energy for the socket and RAM on CPUs, as the power consumption on GPUs or FPGAs also includes the power required by the device RAM. On AMD CPUs we do not have the ability to measure energy consumption of the RAM, so measurements for AMD CPUs only includes CPU cores and caches.

We measure and compare the energy consumption on seven architectures: Nvidia Grace (ARM), Intel Xeon and AMD EPYC (both x86) CPUs, AMD MI210 GPU, Nvidia H100 GPU contained within a Grace-Hopper module, as well as Intel Stratix 10 FPGAs with and without HBM. To compare the energy efficiency across mesh sizes, we plot the measured data as degrees of freedom per joule (DOF/J) in Fig. 8a.

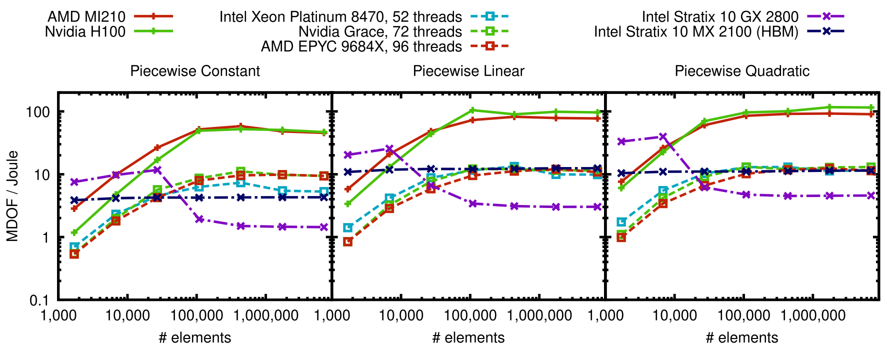
Similarly to the performance shown in Fig. 5, we need relatively large mesh sizes for peak energy efficiency on CPUs and GPUs. While the Intel CPU has a relatively constant power consumption, the CPUs from AMD and Nvidia show some fluctuations in the reported power draw (Fig. 8b). Smaller meshes consume less power than the larger meshes, and there appears to be a peak in the power consumption depending on the architecture and approximation order. Normalized to degrees of freedom per joule (DOF/Joule, Fig. 8b), the AMD and Nvidia CPU yield a better energy efficiency for the piecewise constant approximation order (around 9.4 MDOF/Joule for the Grace CPU and 5.3 MDOF/Joule for the Intel CPU). For GPUs, we observe a similar effect, most notably in the Nvidia H100 GPU on a mesh size of 434,000 elements. The most likely explanation for this peak is a shift in hardware utilization due to global tasks (simulation stability checks) still performed on the host device (CPU) outside of the SYCL code and the time required for this routine increasing with the mesh size. With the 434,000 elements mesh, this takes less than 10% of the overall runtime, but increases to 25 % for the largest mesh, leaving the GPU idling. For the AMD MI210 this effect is less observable due to the lower peak performance achieved by this GPU.

The FPGAs show constant power consumption for all mesh sizes. On the Stratix 10 GX card we observe a high energy efficiency for the smallest meshes. This is a combination of the high throughput when the mesh resides completely inside the cache and the low power consumption. On the HBM card we see energy efficiency comparable to the tested CPUs for P0 and P1, because the 5x difference in performance is compensated by a 5x higher power usage on CPUs.

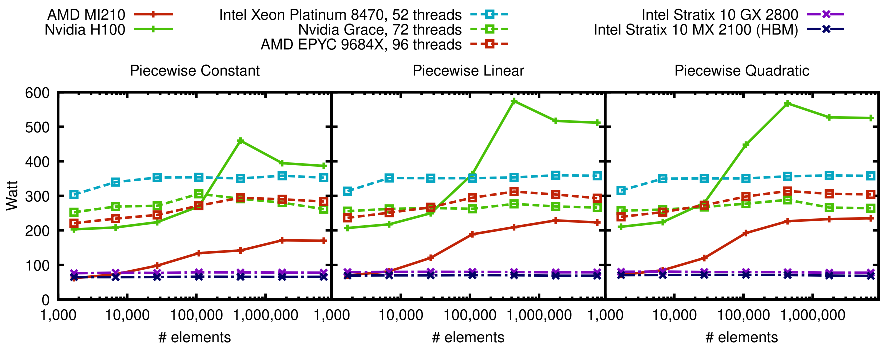
5.7 Portability and performance case studies

In the previous sections, we analyzed performance across CPUs, GPUs, and FPGAs from different vendors. All the tested hardware can be found in current- or next-generation server systems. We will now show that our code is also portable to consumer-grade hardware, e.g., notebooks or desktop workstations.

Table 5 shows the runtime in seconds, energy consumption in joules, and average power draw in watts for a range of systems for a tidal simulation in Galveston Bay. In addition to server processors like the AMD EPYC 9684, tested hardware also includes



(a) Energy efficiency in degrees of freedom per joule for different architectures. Higher values are better.



(b) Average power draw in watt for different mesh sizes and architectures.

Fig. 8 Energy efficiency and power consumption for various mesh sizes and architectures. The energy is only measured for the device running the SYCL code, i.e., CPU, GPU, or FPGA, and does not include energy used by the host system. For CPUs and GPUs, we use AdaptiveCpp 24.02 as compiler with OpenMP backend on AMD and Nvidia CPUs and OpenCL backend on the Intel CPU

workstations (Intel i9-12900K, Apple M1) or notebook processors (Ryzen 3500U). For GPUs, we tested integrated GPUs (UHD Graphics 770, part of the i9-12900K CPU), consumer-grade GPUs (RTX 3070, RX 6900 XT), as well as data-center hardware (H100, L4, MI210). Additionally, we include the Stratix 10 MX (with HBM) and Stratix 10 GX (without HBM) FPGAs.

On all tested architectures, we were able to run our code without further modifications. High-end data center hardware usually has the lowest overall runtime, although we see comparable performance also in some cases on consumer hardware (for example with the AMD MI210 and AMD RX 6900 XT). The Intel Stratix 10 MX FPGA is competing in performance with the Apple M1 CPU in this medium size scenario (2 times slower for P0, 2 times faster for P2), but CPUs like the Intel Xeon Platinum 8470 are faster by a factor of 5 – 10 at P2.

In the energy to solution metric, GPUs are the clear winner for all approximation orders. Notably, the Nvidia L4 GPU requires much less energy compared to other

Table 5 Runtime and energy data for the tidal flow simulation in Galveston Bay (cf. Section 3.3.2 with 54,352 elements)

Device		P0			P1			P2		
		Time	Energy	Power	Time	Energy	Power	Time	Energy	Power
		s	J	W	s	J	W	s	J	W
2023, S	Intel Xeon 8470	*3.6*	**1241.2**	***344.8***	*22.5*	**7902.6**	***351.2***	*124.7*	**43,682.9**	***350.3***
2023, S	Nvidia Grace	*3.2*	**702.5**	***219.5***	*29.2*	**7673.2**	***262.8***	*148.8*	**39,471.0**	***265.3***
2021, S	Ampere Altra M128-30	*4.6*	**195.0**	***42.4***	*30.8*	**5579.9**	***181.2***	*220.0*	**45,364.8**	***206.2***
2023, S	AMD EPYC 9684X	*4.6*	**1137.2**	***247.6***	*38.9*	**10,393.3**	***267.2***	207.7	**58,271.0**	***280.5***
2021, C	Intel i9-12900K	*8.5*	**1793.7**	***211.0***	*59.9*	**10,750.8**	***179.5***	*329.0*	**44,584.9**	***135.5***
2021, S	Intel Xeon 8358	*11.2*	**1637.2**	***146.2***	*63.7*	**10,185.2**	***159.9***	*301.6*	**50,500.9**	***167.4***
2021, C	Apple M1 Studio	*18.8*	–	–	*173.6*	–	–	*1335.4*	–	–
2019, C	AMD Ryzen 3500U	*66.9*	**741.4**	***11.1***	*563.0*	**5992.7**	***10.6***	*3110.1*	**30,740.3**	***9.9***
2023, S	Nvidia H100	*1.1*	**244.7**	***222.5***	*4.4*	**1281.1**	***291.2***	*15.3*	**5095.4**	***333.0***
2023, S	Nvidia L4	*1.6*	**72.6**	***45.4***	*5.9*	**411.8**	***69.8***	*28.2*	**1975.3**	***70.0***
2020, C	AMD RX 6900 XT	*2.7*	**200.2**	***74.1***	*8.3*	**1442.3**	***173.8***	*29.3*	**6954.5**	***237.4***
2022, S	AMD MI210	*2.0*	**149.1**	***74.6***	*10.5*	**1327.3**	***126.4***	*46.9*	**7255.7**	***154.7***
2020, C	Nvidia RTX 3070	*2.0*	**181.2**	***90.6***	*11.3*	**1732.4**	***152.8***	*58.3*	**10,102.2**	***173.3***
2021, C	Intel UHD 770	*8.6*	–	–	*92.7*	–	–	*852.5*	–	–
2020, S	Stratix 10 MX 2100	*25.8*	**1703.1**	***66.1***	*101.8*	**7191.4**	***70.6***	*652.6*	**46,849.4**	***71.8***
2018, S	Stratix 10 GX 2800	*29.6*	**2343.7**	***79.2***	*262.1*	**20,885.9**	***79.7***	*1215.5*	**94,122.0**	***77.4***

Energy is given in joules and power in watts. Energy consumption between different systems is not directly comparable because it is measured at different hardware locations and may include different components. For the Apple M1 CPU and Intel UHD Graphics 770 we were not able to measure energy usage. Consumer-grade hardware is marked with a 'C' after the release year, server hardware with an 'S'

Table 6 Runtime and energy data for Bahamas mesh (see Sect. 3.3.1) with 27,136 elements

Device	P0			P1			P2		
	Time	Energy	$\frac{\text{MDOF}}{\text{J}}$	Time	Energy	$\frac{\text{MDOF}}{\text{J}}$	Time	Energy	$\frac{\text{MDOF}}{\text{J}}$
	s	J		s	J		s	J	
Intel Xeon 8470	*1.1*	**304.5**	***4.62***	*6.4*	**1934.2**	***8.73***	*16.6*	**5373.1**	***9.43***
Nvidia Grace	*0.9*	**287.4**	***4.89***	*8.6*	**1960.5**	***8.61***	*20.9*	**5739.5**	***8.82***
AMD EPYC 9684X	*1.4*	**322.4**	***4.36***	*10.6*	**2676.9**	***6.31***	*28.3*	**7363.9**	***6.88***
Nvidia H100	*0.5*	**97.0**	***14.51***	*1.8*	**425.1**	***39.70***	*2.9*	**781.5**	***64.80***
AMD MI210	*0.6*	**28.7**	***48.96***	*3.1*	**255.2**	***66.14***	*7.4*	**791.9**	***63.95***
Stratix 10 MX 2100	*5.1*	**317.5**	***4.43***	*19.9*	**1386.6**	***12.20***	*64.3*	**4548.1**	***11.13***
Stratix 10 GX 2800	*1.6*	**118.1**	***11.82***	*31.7*	**2537.6**	***6.66***	*102.9*	**8196.4**	***6.17***

Simulation time is one day with a time step of 5 s (P0) or 2.5 s (P1, P2)

GPUs at a similar runtime. An Nvidia L4 GPU has a TDP of only 72 watts, a memory bandwidth of 300 GB/s, and an L2 cache size of 48 MB. With the tested mesh size of 54,352 elements, the entire mesh information and degrees of freedom fit into the L2 cache, and the lower memory bandwidth is less important for the overall performance.

Table 6 shows the wall clock time, energy in joules, and energy efficiency in million degrees of freedom per joule (MDOF/J) for the Bahamas mesh with 27,136 elements. For piecewise constant approximation (P0), the mesh fits entirely into the FPGA caches on the Stratix 10 GX FPGA, and hence we see much better performance compared to the Stratix 10 MX. The required time to solution is then close to the CPUs with a lower energy cost. On the remaining architectures, results are consistent with results from the Galveston Bay simulation: Again, GPUs have the lowest energy requirements, but, for the smaller mesh used in this setup, the runtime difference between CPUs and GPUs is lower. This is an expected outcome as the GPUs are underutilized with small meshes.

6 Conclusions and outlook

In this work, we used SYCL to implement a 2D shallow water solver, discretized on unstructured triangular meshes via the discontinuous Galerkin method. Through the combination of SYCL's platform portable programming model and C++ templates, we can reuse the numerical algorithm on all architectures. To improve performance portability, data layouts and accesses have been separated from the numerical algorithm, and a separate code path was created for FPGAs due to their different execution model. The resulting code has been tested on a wide variety of hardware, including x86 and ARM CPUs, consumer and data-center GPUs, and two different Intel Stratix 10 FPGAs. With the discussed specializations, we have qualitatively reached the goal of reasonably high application performance in relation to the different platform capabilities. However, our results also show limitations in performance portability, both regarding the performance differences caused by the compiler and backend alternatives as well as due to trade-offs within the code itself.

Comparing the resulting performance and energy efficiency on different target platforms, we observed that Intel Stratix 10 MX 2100 FPGAs with HBM achieve twice the performance of the FPGA without HBM for the larger meshes, but the caches on the Intel Stratix 10 GX 2800 FPGA still performed best for the smallest mesh sizes yielding the highest performance across all tested architectures. For all larger meshes, GPUs reached the highest application performance and, compared to current-generation server CPUs, had up to ten times higher energy efficiency. FPGAs had comparable, or for the small meshes, higher energy efficiency than the CPUs.

While the current work focuses on the shallow water equations, the optimizations and parallelization strategies should be transferable to other applications as well. A key component here is parallelization over the elements: The explicit time discretization allows a trivial parallelization over elements, as each triangle has its own set of degrees of freedom. For (semi-)implicit discretizations, or in the presence of linear solvers, parallelization might not be as straightforward.

There are several directions to pursue for further performance enhancements, but it remains to be seen how far this is possible with an approach maintaining a possibly high degree of performance portability. Hierarchical parallelism and task graphs are features that are on the roadmap of the SYCL standard and could help to improve locality in cache hierarchies and mitigate limitations of global synchronization points, as for example demonstrated for an electromagnetic DG code in [54]. Separate partitions of elements with communicating halos can also be used to add further parallelism to the FPGA design, as demonstrated in the target-specific approach in [37], and to scale over multiple devices of any architecture. This future work might involve additional trade-offs between computational efficiency for a specific subset of hardware and performance portability, particularly with regard to typical MPI communication versus direct streaming interfaces between FPGAs. Moreover, in strong scaling scenarios, the ability of architectures to approach their peak performance for smaller mesh sizes will become more relevant.

Finally, it would be interesting to implement block-structured grids and the quadrature-free reformulation as presented in [35] allowing us to directly compare the performance with the automatically generated code of ExaStencils. Block-structured grids have the potential to further speed up simulations by utilizing, for example, shared memory on GPUs to reduce the amount of data loaded from the DRAM. The quadrature-free formulation contains an auxiliary variable for the depth-averaged velocities, which is the solution of a small linear system that needs to be solved on each triangle. On the other hand, the analytic evaluation of the integrals could also reduce the amount of required computations (if they are shifted to the compilation phase).

As far as applications are concerned, additional modules like wetting and drying, sedimentation, or species transport could be integrated into this codebase. Performance implications of each of these extensions are hard to predict and depend on the specific model used. With a wetting and drying scheme one has to deal with elements becoming active or inactive, which could be challenging for load-balancing.

Appendix A Kernel launch codes

Listing 1: Kernel launch code for CPUs and GPUs. The block size of 256 elements was chosen empirically, but works reasonably well across all mesh sizes and architectures.

```
for (int step = 0; step < nsteps; step++) {
  for (int rk = 0; rk < d_rk; rk++) {
    queue.submit([&](auto& handler) {
      // omitted: get accessors
      handler.parallel_for(sycl::nd_range<1>{{num_elems}, {256}},
        [=](sycl::nd_item<1> id) {
          process_element(id, /* other arguments */);
        });
    });
  }
}
```

Listing 2: Kernel launch code for FPGAs

```
queue.submit([&](auto& handler) {
  // omitted: get accessors
  handler.single_task([=]() {
    // omitted: prepare data caches
    for (int step = 0; step < nsteps; step++) {
      for (int rk = 0; rk < d_rk; rk++) {
        for (unsigned id = 0; id < num_elems; id++) {
          process_element(id, /* other arguments */);
        }
      }

      t += dt;
    }
  });
});
```

Appendix B Hardware parameters

See Table 7.

Table 7 Parameters for the tested CPUs and GPUs in Fig. 5

CPU	# cores	Memory bandwidth (GB/s)	GFLOPS	
Intel Xeon Platinum 8470	52	262	206	Scalar
			2402	AVX
			3,145	AVX-512
AMD EPYC 9684X	96	373	625	Scalar
			6422	AVX
			6,627	AVX-512
Nvidia Grace	72	445	811	Scalar
			6488	SVE-256
			13,250	SVE-512

GPU	# SM/CU	Memory bandwidth (TB/s)	GFLOPS
AMD MI210	104	1.6	22,630
Nvidia H100	132	4.0	50,677

Peak memory bandwidth and GFLOPS on the CPUs were measured using LIKWID

Acknowledgements This research was funded by the German Research Foundation (DFG) under grants AI 117/7-1 and KE 2844/1-1 within the project 502500606 'Performance-optimiertes Co-Design von Ozeanmodellierungssoftware auf FPGAs.' The authors gratefully acknowledge the scientific support and HPC resources provided by the Erlangen National High Performance Computing Center (NHR@FAU) of the Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU). The hardware is funded by the German Research Foundation (DFG). The authors gratefully acknowledge the computing time provided to them on the high-performance computer Noctua 2 at the NHR Center PC2. These are funded by the German Federal Ministry of Education and Research and the state governments participating on the basis of the resolutions of the GWK for the national high-performance computing at universities (www.nhr-verein.de/unsere-partner). We gratefully acknowledge the support given through the Competence Network for Scientific High Performance Computing in Bavaria (KONWIHR).

Author contributions Conceptualization: Markus Büttner, Christoph Alt, Tobias Kenter, Vadym Aizinger; Software: Markus Büttner, Christoph Alt; Writing: Markus Büttner, Christoph Alt, Tobias Kenter, Vadym Aizinger; Visualization: Markus Büttner, Christoph Alt; Funding acquisition: Tobias Kenter, Vadym Aizinger; Supervision: Tobias Kenter, Harald Köstler, Christian Plessl, Vadym Aizinger; Project administration: Vadym Aizinger

Funding Open Access funding enabled and organized by Projekt DEAL.

Data availablity Data sets generated during the current study are available from the corresponding author on reasonable request.

Declarations

Competing interests The authors declare no competing interests.

References

1. Pennycook SJ, Sewall JD, Jacobsen DW, Deakin T, McIntosh-Smith S (2021) Navigating performance, portability, and productivity. Computi Sci Eng 23(5):28–38. https://doi.org/10.1109/MCSE.2021.3097276
2. Pennycook SJ, Sewall JD (2021) Revisiting a metric for performance portability. In: 2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC), pp. 1–9 . https://doi.org/10.1109/P3HPC54578.2021.00004
3. Marowka A (2022) Reformulation of the performance portability metric. Softw Pract Exp 52(1):154–171. https://doi.org/10.1002/spe.3002
4. Marowka A (2023) A comparison of two performance portability metrics. Concurr Comput Pract Exp 35(25):7868. https://doi.org/10.1002/cpe.7868
5. Trott CR, Lebrun-Grandié D, Arndt D, Ciesko J, Dang V, Ellingwood N, Gayatri R, Harvey E, Hollman DS, Ibanez D, Liber N, Madsen J, Miles J, Poliakoff D, Powell A, Rajamanickam S, Simberg M, Sunderland D, Turcksin B, Wilke J (2022) Kokkos 3: Programming model extensions for the exascale era. IEEE Trans Parallel Distrib Syst (TPDS) 33(4):805–817. https://doi.org/10.1109/TPDS.2021.3097283
6. Deakin T, McIntosh-Smith S (2020) Evaluating the performance of HPC-style SYCL applications. In: Proceedings of the International Workshop on OpenCL. IWOCL '20, pp. 1–11. Association for Computing Machinery, New York, NY, USA . https://doi.org/10.1145/3388333.3388643
7. Reguly IZ (2023) Evaluating the performance portability of SYCL across CPUs and GPUs on bandwidth-bound applications. In: Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis. SC-W '23, pp. 1038–1047. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3624062.3624180
8. Apanasevich L, Kale Y, Sharma H, Sokovic AM (2024) A Comparison of the performance of the molecular dynamics simulation package gromacs implemented in the sycl and cuda programming models https://doi.org/10.48550/ARXIV.2406.10362 . Publisher: arXiv, Version Number: 1. Accessed 2025-01-09
9. Rangel EM, Pennycook SJ, Pope A, Frontiere N, Ma Z, Madananth V (2023) A Performance-Portable SYCL Implementation of CRK-HACC for Exascale. In: Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, pp. 1114–1125. ACM, Denver CO USA. https://doi.org/10.1145/3624062.3624187
10. Nichols NS, Childers JT, Burch TJ, Field L (2024) Improving Performance Portability of the Procedurally Generated High Energy Physics Event Generator MadGraph Using SYCL. In: Proceedings of the 12th International Workshop on OpenCL And SYCL. IWOCL '24, pp. 1–8. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3648115.3648116
11. Alpay A, Heuveline V (2023) One pass to bind them: The first single-pass sycl compiler with unified code representation across backends. In: Proc Int Workshop on OpenCL (IWOCL). IWOCL '23. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3585341.3585351
12. Meyer J, Alpay A, Hack S, Fröning H, Heuveline V (2023) Implementation techniques for SPMD kernels on CPUs. In: Proc Int Workshop on OpenCL (IWOCL). IWOCL '23. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3585341.3585342
13. Büttner M, Alt C, Kenter T, Köstler H, Plessl C, Aizinger V (2024) Enabling Performance Portability for Shallow Water Equations on CPUs, GPUs, and FPGAs with SYCL. In: Proceedings of the platform for advanced scientific computing conference. ACM, Zurich Switzerland, pp 1–12. https://doi.org/10.1145/3659914.3659925
14. Páll S, Zhmurov A, Bauer P, Abraham M, Lundborg M, Gray A, Hess B, Lindahl E (2020) Heterogeneous parallelization and acceleration of molecular dynamics simulations in GROMACS. J Chem Phys 153(134110):1–15. https://doi.org/10.1063/5.0018516

15. Dufek AS, Gayatri R, Mehta N, Doerfler D, Cook B, Ghadar Y, DeTar C (2021) Case study of using kokkos and sycl as performance-portable frameworks for milc-dslash benchmark on nvidia, amd and Intel gpus. In: 2021 International workshop on performance, portability and productivity in HPC (P3HPC), pp 57–67. https://doi.org/10.1109/P3HPC54578.2021.00009
16. Alpay A, Heuveline V (2020) SYCL beyond opencl: The architecture, current state and future direction of hipSYCL. In: Proc Int Workshop on OpenCL and SYCL (IWOCL). IWOCL '20. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3388333.3388658
17. Pennycook SJ, Sewall JD, Lee VW (2016) A metric for performance portability. https://arxiv.org/abs/1611.07409
18. Pennycook SJ, Sewall JD, Lee VW (2019) Implications of a metric for performance portability. Future Gener Comput Syst 92:947–958. https://doi.org/10.1016/j.future.2017.08.007
19. Marowka A (2024) Portability efficiency approach for calculating performance portability . https://arxiv.org/abs/2407.00232
20. Weckert C, Solis-Vasquez L, Oppermann J, Koch A, Sinnen O (2023) Altis-SYCL: migrating altis benchmarking suite from CUDA to SYCL for GPUs and FPGAs. In: Proc workshop on heterogeneous high-performance reconfigurable computing (H2RC), Held in Conjuction with Int Conf on High Performance Computing, Networking, Storage and Analysis (SC). SC-W '23. Association for Computing Machinery, New York, NY, USA, pp 547–555. https://doi.org/10.1145/3624062.3624542
21. Hu B, Rossbach CJ (2020) Altis: Modernizing GPGPU benchmarks. In: Proc IEEE Int Symp on Performance Analysis of Systems and Software (ISPASS), pp 1–11. https://doi.org/10.1109/ISPASS48437.2020.00011
22. Wang Y, Zhou Y, Wang QS, Wang Y, Xu Q, Wang C, Peng B, Zhu Z, Takuya K, Wang D (2021) Developing medical ultrasound beamforming application on GPU and FPGA using oneAPI. In: Proc Int Symp on Parallel and Distributed Processing Workshops (IPDPSW), pp. 360–370 . https://doi.org/10.1109/IPDPSW52791.2021.00064
23. Kenter T, Shambhu A, Faghih-Naini S, Aizinger V (2021) Algorithm-hardware co-design of a discontinuous Galerkin shallow-water model for a dataflow architecture on FPGA. In: Proceedings of the Platform for Advanced Scientific Computing Conference, pp. 1–11. ACM, Geneva Switzerland . https://doi.org/10.1145/3468267.3470617
24. Wu X, Kenter T, Schade R, Kühne TD, Plessl C (2023) Computing and compressing electron repulsion integrals on FPGAs. In: Proc IEEE Symp on Field-Programmable Custom Computing Machines (FCCM), pp 162–173. https://doi.org/10.1109/FCCM57271.2023.00026
25. Opdenhövel J-O, Plessl C, Kenter T (2023) Mutation tree reconstruction of tumor cells on FPGAs using a bit-level matrix representation. In: Proc Int Symp Highly-Efficient Accelerators and Reconfigurable Technologies (HEART). HEART '23, pp. 27–34. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3597031.3597050
26. Olgu K, Kenter T, Nunez-Yanez J, Mcintosh-Smith S (2024) Optimisation and evaluation of breadth first search with oneAPI/SYCL on Intel FPGAs: from describing algorithms to describing architectures. In: Proc Int Workshop on OpenCL and SYCL (IWOCL). IWOCL '24. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3648115.3648134
27. Opdenhövel J-O, Alt C, Plessl C, Kenter T (2024) Stencilstream: A SYCL-based stencil simulation framework targeting FPGAs. In: Proc Int Conf on Field Programmable Logic and Applications (FPL), pp. 100–108. https://doi.org/10.1109/FPL64840.2024.00023
28. Gandham R, Medina D, Warburton T (2015) GPU accelerated discontinuous Galerkin methods for shallow water equations. Commun Comput Phys 18(1):37–64. https://doi.org/10.4208/cicp.070114.271114a
29. Caviedes-Voullième D, Morales-Hernández M, Norman MR, Özgen-Xian I (2023) SERGHEI (SERGHEI-SWE) v1.0: a performance-portable high-performance parallel-computing shallow-water solver for hydrology and environmental hydraulics. Geosci Model Dev 16(3):977–1008. https://doi.org/10.5194/gmd-16-977-2023
30. Aureli F, Prost F, Vacondio R, Dazzi S, Ferrari A (2020) A GPU-accelerated shallow-water scheme for surface runoff simulations. Water 12(3):637. https://doi.org/10.3390/w12030637
31. Morales-Hernández M, Sharif MB, Kalyanapu A, Ghafoor SK, Dullo TT, Gangrade S, Kao S-C, Norman MR, Evans KJ (2021) TRITON: a multi-GPU open source 2D hydrodynamic flood model. Environ Modell Softw 141:105034. https://doi.org/10.1016/j.envsoft.2021.105034
32. Reinarz A, Charrier DE, Bader M, Bovard L, Dumbser M, Duru K, Fambri F, Gabriel A-A, Gallard J-M, Köppel S, Krenz L, Rannabauer L, Rezzolla L, Samfass P, Tavelli M, Weinzierl T (2020) ExaHyPE:

an engine for parallel dynamically adaptive simulations of wave problems. Comput Phys Commun 254:107251. https://doi.org/10.1016/j.cpc.2020.107251

33. Loi CM, Bockhorst H, Weinzierl T (2024) SYCL compute kernels for ExaHyPE, pp. 90–103 . https://doi.org/10.1137/1.9781611977967.8
34. Hélène C, Minh-Hoang L, Sébastien L (2013) Parallelization of shallow-water equations with the algorithmic skeleton library SkelGIS. Procedia Computer Science 18:591–600 https://doi.org/10.1016/j.procs.2013.05.223 . 2013 International Conference on Computational Science
35. Faghih-Naini S, Kuckuk S, Zint D, Kemmler S, Köstler H, Aizinger V (2023) Discontinuous Galerkin method for the shallow water equations on complex domains using masked block-structured grids. Advances in Water Resources 182, 104584 https://doi.org/10.1016/j.advwatres.2023.104584
36. Alt C, Kenter T, Faghih-Naini S, Faj J, Opdenhövel J-O, Plessl C, Aizinger V, Hönig J, Köstler H (2023) Shallow water DG simulations on FPGAs: Design and comparison of a novel code generation pipeline. In: Proc Int Conf on High Performance Computing (ISC High Performance). Lecture Notes in Computer Science (LNCS), pp. 86–105. Springer, Cham . https://doi.org/10.1007/978-3-031-32041-5_5
37. Faj J, Plessl C, Kenter T, Faghih-Naini S, Aizinger V (2023) Scalable Multi-FPGA Design of a Discontinuous Galerkin Shallow-Water Model on Unstructured Meshes. In: Proc Platform for Advanced Scientific Computing Conf (PASC), pp. 1–12 .https://doi.org/10.1145/3592979.3593407
38. Kenter T, Mahale G, Alhaddad S, Grynko Y, Schmitt C, Afzal A, Hannig F, Förstner J, Plessl C (2018) OpenCL-based FPGA design to accelerate the nodal discontinuous Galerkin method for unstructured meshes. In: Proc IEEE Symp on Field-Programmable Custom Computing Machines (FCCM), pp. 189–196. IEEE, New York, NY, USA . https://doi.org/10.1109/FCCM.2018.00037
39. Gourounas D, Hanindhito B, Fathi A, Trenev D, John LK, Gerstlauer A (2023) FAWS: Fpga acceleration of large-scale wave simulations. In: Proc IEEE Int Conf on Application-Specific Systems, Architectures, and Processors (ASAP), pp. 76–84 . https://doi.org/10.1109/ASAP57973.2023.00025
40. Düben PD, Korn P, Aizinger V (2012) A discontinuous/continuous low order finite element shallow water model on the sphere. J Comput Phys 231(6):2396–2413. https://doi.org/10.1016/j.jcp.2011.11.018
41. Aizinger V, Dawson C (2002) A discontinuous Galerkin method for two-dimensional flow and transport in shallow water. Adv Water Resour 25(1):67–84. https://doi.org/10.1016/S0309-1708(01)00019-7
42. Hajduk H, Hodges BR, Aizinger V, Reuter B (2018) Locally filtered transport for computational efficiency in multi-component advection-reaction models. Environ Model Softw 102:185–198. https://doi.org/10.1016/j.envsoft.2018.01.003
43. Cockburn B, Shu C-W (1989) TVB Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws. II. General framework Math Comp 52:411–435. https://doi.org/10.1090/S0025-5718-1989-0983311-4
44. Chippada S, Dawson CN, Martinez ML, Wheeler MF (1998) A Godunov-type finite volume method for the system of shallow water equations. Comput Methods Appl Mech Eng 151(1):105–129. https://doi.org/10.1016/S0045-7825(97)00108-4
45. Faghih-Naini S, Kuckuk S, Aizinger V, Zint D, Grosso R, Köstler H (2020) Quadrature-free discontinuous Galerkin method with code generation features for shallow water equations on automatically generated block-structured meshes. Adv Water Resour 138:103552
46. Westerink JJ, Stolzenbach KD, Connor JJ (1989) General spectral computations of the nonlinear shallow water tidal interactions within the bight of Abaco. J Phys Oceanogr 19(9):1348–1371
47. Sakiotis I, Arumugam K, Paterno M, Ranjan D, Terzić B, Zubair M (2023) Porting Numerical Integration Codes from CUDA to oneAPI: a case study. In: Bhatele, A., Hammond, J., Baboulin, M., Kruse, C. (eds.) High Performance Computing, pp. 339–358. Springer, Cham . https://doi.org/10.1007/978-3-031-32041-5_18
48. Bauer C, Kenter T, Lass M, Mazur L, Meyer M, Nitsche H, Riebler H, Schade R, Schwarz M, Winnwa N, Wiens A, Wu X, Plessl C, Simon J (2024) Noctua 2 supercomputer. J Large-scale Res Facil (JLSRF). https://doi.org/10.17815/jlsrf-8-187
49. Meyer M, Kenter T, Plessl C (2022) In-depth FPGA accelerator performance evaluation with single node benchmarks from the HPC challenge benchmark suite for Intel and Xilinx FPGAs using OpenCL. J Parallel Distrib Comput 160:79–89. https://doi.org/10.1016/j.jpdc.2021.10.007
50. Gruber, T., Eitzinger, J., Hager, G., Wellein, G.: LIKWID. https://doi.org/10.5281/zenodo.10105559
51. Schöne R, Ilsche T, Bielert M, Velten M, Schmidl M, Hackenberg D (2021) Energy efficiency aspects of the AMD Zen 2 architecture. In: 2021 IEEE International Conference on Cluster Computing (CLUSTER), pp. 562–571. IEEE, Portland, OR, USA . https://doi.org/10.1109/Cluster48925.2021.00087

52. NVIDIA: NVIDIA Grace Performance Tuning Guide. (2024). https://docs.nvidia.com/grace-performance-tuning-guide.pdf Accessed 2024-06-19
53. Corda, S., Veenboer, B., Tolley, E.: PMT: power measurement toolkit. In: 2022 IEEE/ACM International Workshop on HPC User Support Tools (HUST), pp. 44–47 (2022). https://doi.org/10.1109/HUST56722.2022.00011
54. Alhaddad, S., Förstner, J., Groth, S., Grünewald, D., Grynko, Y., Hannig, F., Kenter, T., Pfreundt, F.-J., Plessl, C., Schotte, M., Steinke, T., Teich, J., Weiser, M., Wende, F.: The HighPerMeshes framework for numerical algorithms on unstructured grids. Concurrency and Computation: Practice and Experience 34, 1–15 (2021) https://doi.org/10.1002/cpe.6616